IDENTIFYING AND MITIGATING THREATS FROM EMBEDDING THIRD-PARTY CONTENT

A Dissertation
Presented to
The Academic Faculty

By

Wei Meng

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2017

IDENTIFYING AND MITIGATING THREATS FROM EMBEDDING THIRD-PARTY CONTENT

Approved by:

Dr. Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Dr. Taesoo Kim
School of Computer Science
Georgia Institute of Technology

Dr. Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara

Dr. Nick Feamster
Department of Computer Science
Princeton University

Date Approved: July 20, 2017

To my parents,

and those who have supported me.

ACKNOWLEDGEMENTS

This Ph.D. dissertation would not exist without the support from a number of people. I would like to take this opportunity to acknowledge them.

First of all, I am very grateful to my advisor, Wenke Lee, for his guidance and support through the Ph.D. program. Wenke has been an amazing advisor, who has trained me to become an independent researcher and think critically. When I first started my Ph.D. I was a little bit disappointed that I did not receive much instructions on projects or concrete research topics from him. Instead, Wenke provided me with the freedom and necessary resources to explore my own interests. When I met difficulties in exploring new directions, his insightful feedback helped me overcome many challenges. I learned what top-quality research is through the many debates with him, which I enjoyed a lot. Wenke has also influenced me on my judgement of what is important in life. I shall benefit from his advice in the rest of my career and life.

I would also like to thank Mustaque Ahamad, Giovanni Vigna, Taesoo Kim and Nick Feamster, for serving on my dissertation committee. Their valuable suggestions and comments helped me make significant improvement to this dissertation.

The work presented in this dissertation would not be possible without the help from my dear collaborators. In my Ph.D., I have been fortunate for having worked with the following brilliant people: Xinyu Xing, Anmol Sheth, Udi Weinsberg, Byoungyoung Lee, Simon P. Chung, Sangho Lee, Christopher Kruegel, Roberto Perdisci, Ren Ding, and Steven Han. Many experiments carried out in this work were supported by the administration team of the Institute for Information Security & Privacy at Georgia Tech, led by Paul Royal and Adam Allred. I am very appreciative of their professional support.

Finally, I must thank my parents, for their unconditional love and support. They have always been providing me with the freedom and support for pursuing my dreams. I shall be able to overcome any challenge in the rest of my life with them standing by me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Embedding content from third parties to enrich features is a common practice in the development of modern web applications and mobile applications. For example, functionalities such as social integration, programming enhancement, visitor tracking and advertisements can be provided by including simple code snippets from third parties. Such practices can pose serious security and privacy threats to an end user, because sensitive data about a user in an application can be *directly* accessed by third-party content that usually operates with the same privilege as first-party content. Third parties can often abuse their privilege to compromise the confidentiality and integrity of the hosting applications and harm end users. The confidentiality and integrity of a user's *indirect* data, such as a user profile, may also be compromised by such practices.

This dissertation aims to identify new threats posed to end users by the practices of embedding third-party content and develop techniques to mitigate these threats. We first demonstrate how a malicious first-party application can pollute a user's indirect data in a third-party service or application by embedding it. In particular, we show that the personalization systems of popular websites can be abused by a malicious publisher to pollute a user's profile in a third-party website. In addition, we demonstrate that a malicious Android application can infer a user's profile that is already learned by a third-party ad network. The pollution and inference of a user's indirect data in other applications are two new classes of threats to end users. We then propose defense techniques to mitigate these two new classes of threats.

This dissertation also aims to design mechanisms that enable end users and developers to limit the privilege of third-party content to prevent unintended behaviors. First, third-party content (*e.g.*, trackers) has the ability to link a user's activities across multiple applications and then build a profile of the user to serve personalized content. Third-party tracking often happens without the user's consent and has raised serious privacy concerns. In this

dissertation, we present TrackMeOrNot, a client-side tracking control mechanism that allows end users to selectively opt out of third-party web tracking based on their demand. Second, existing web security mechanisms offer all-or-nothing restriction on the privilege of third-party content, *e.g.*, embedded JavaScript code. Most developers have to trade security for the benefit of embedding third-party JavaScript code, making these mechanisms ineffective. To this end, we study how third-party JavaScript code accesses a user's direct data in a web application in general through a large-scale measurement. Our results show that third-party JavaScript code is over-privileged and such privilege is abused to access sensitive user data. To address the limitations of existing web permission mechanisms, we propose the fine-grained Access Control Policy for the Document Object Model (`DOM-ACP`). `DOM-ACP` can help web developers restrict the privilege of third-party JavaScript code in their applications. The new mechanism enables web developers to specify the access permission of a third-party script at per-object granularity to deny unauthorized accesses and protect the confidentiality and integrity of sensitive content in a web application.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

Modern web applications[1] and mobile applications extensively collect data about their end users to improve user experience. Data associated with a user can be categorized into two classes: *direct data* and *indirect data*. Direct data is any personal or personalized data that is directly presented (accessible) to a user in an application. For example, an application may display the user name or email address in its interface after a user has been authenticated. On the other hand, indirect data such as the recently browsed items of a user, is gathered and stored in the back end servers of an application, which uses the collected indirect data to provide end users with better and personalized services. For instance, by building a user profile from the collected indirect data, an application can tailor content to be more relevant to a specific user. The compiled user profile is also one type of indirect user data.

In the development of modern web applications and mobile applications, embedding content served by other providers to enrich features is a common practice. For instance, functionalities such as programming enhancement (*e.g.*, jQuery), social integration (*e.g.*, widgets of Facebook and Twitter), visitor tracking (*e.g.*, Google Analytics), and advertising (*e.g.*, Google DoubleClick) can be easily implemented by embedding a simple code snippet provided by a third party into one's own application. Many users are not aware of and/or can not recognize the presence of third-party content. As a result, an end user may unintentionally interact with third-party content in addition to the application that she or he currently uses. While interacting with an application, data about an end user is collected by the application the user directly interacts with, as well as the third-party content that is embedded within

[1]Throughout this thesis, we use *websites* and *web applications*, as well as *JavaScript* and *script* interchangeably.

the application. In many cases, such data is collected and used in ways that are out of the user's control and without the user's consent.

Third-party content can pose serious security and privacy threats to an application and its users, because it usually operates at full privilege as the embedding application. Sensitive user data, *e.g.*, user credentials, billing information and user inputs, can be directly accessed by third-party content if the data is not well isolated. A malicious third party can even modify the hosting application to cause unintended behavior. Web developers can rely on modern web browsers to isolate third-party content and limit its permission if they embed third-party content (*e.g.*, a third-party web page) within an *iframe* tag, because the Same-Origin Policy (SOP) prohibits the interaction between content loaded from different origins[2]. However, a lot of third-party content such as JavaScript code and Cascading Style Sheet (CSS) can be directly embedded within a hosting application, which is exempt from the restriction imposed by SOP and hence violates fundamental security principles. In the Android mobile operating system, third-party content is not well isolated from the hosting application either and can inherit all the privilege of the hosting application.

It is well known that third parties can often abuse their privilege to compromise the confidentiality and integrity of the hosting applications and harm end users by stealing or modifying direct user data. However, the problem that whether a user's indirect data can be compromised by a malicious third party or even a first party is not well explored. Moreover, neither the end users nor the developers can easily limit the privilege of third-party content to prohibit unintended behaviors.

1.2 Dissertation Overview

This dissertation aims to *identify new threats posed to end users by the practices of embedding third-party content in modern web applications and mobile applications*, and *develop techniques to mitigate these threats*. Furthermore, we aim to *advance the state-of-the-art on*

[2]A web origin is defined as the tuple of protocol, host and port number. Cross-origin request can still be allowed with the Cross-origin resource sharing (CORS) mechanism.

defense techniques against over-privileged third-party content for preventing unintended behaviors.

In the first part of this dissertation (§2 and §3), we study the security of indirect user profiles in applications of popular service providers. Specifically, we have identified and proposed mitigation techniques for two classes of new threats that are posed by a malicious *first-party* application to a user's indirect data. A malicious first-party application can either *pollute* or *infer* a user's profile on another application by embedding it. The second part (§4) presents `TrackMeOrNot`, a system that allows end users to selectively opt out of unwanted third-party web tracking. In the last two chapters (§5 and §6), we study the threats posed to a user's direct data in a web application by third-party JavaScript code. We first conduct a large-scale measurement with `DOM-Logger`, a system that helps web developers monitor and understand the web content access behavior of embedded JavaScript code. We then design `DOM-ACP`, a fine-grained permission mechanism for web applications, as the last contribution of this dissertation. We introduce each of the studied problems and the research approaches taken in the following subsections.

1.2.1 Pollution Attacks

Personalization is an emerging technology that is embraced by modern applications. For example, personalization has been widely used in recommending search results (Google), videos (YouTube, Netflix, *etc.*), items (Amazon) and targeting certain users in online advertising (DoubleClick). Serving more relevant content not only improves user experience and increases user engagement, but can also help the growth in a business's revenue.

To customize content based on a user's interest, gender, age and other personal traits, an application needs to first collect a user's past interactions with it and then build a user profile based on the collected historical indirect data. However, many applications do not properly validate the collected user interaction data. Furthermore, these web applications can be embedded by other web applications as third-party content, thus allowing a malicious

application to trick a user's browser to load the content of embedded applications. The above observations present a new attack surface for attackers to interfere with an application's personalization system and compromise the integrity of a user's indirect data.

In §2, we study whether the personalization systems of popular web applications can be abused by a malicious publisher website. In particular, we experimented with popular web applications that a user interacts with either directly or indirectly. We attempted to inject artificial user interaction data to the personalization systems of these applications by either embedding content of them or exploiting vulnerabilities such as Cross-site Request Forgery (CSRF). We observe that a malicious first-party application was able to alter the customized content of these applications because the input to their personalization systems was polluted. We have proposed several techniques such as browsing context identification to mitigate such threats.

1.2.2 Inferring User Profile with Personalized Mobile In-App Ads

Tracking and advertising libraries are one class of the most commonly embedded third-party content in today's web applications and mobile applications. By collaborating with thousands or even millions of applications, the providers of these third-party libraries have the ability to link a user's activities (indirect data) across multiple applications and build an accurate profile of a user. As a result, the personalized ads are highly correlated with a user's personal information such as her or his interests and demographic information.

In web applications, the personalized ads are usually isolated in iframes from other content by the Same-Origin Policy (SOP). In today's Android applications, such isolation does not exist. Much effort has been put to understanding and mitigating the security and privacy threats posed by embedded third-party content on Android. Little has focused on studying whether a malicious hosting application can also put threats to user privacy.

In §3, we estimate how much a mobile app can learn about a user by abusing its privilege to inspect the personalized ads loaded in app runtime. In particular, we sought the ground

truth regarding users' demographic and interest profile and personalized ads served by Google AdMob from more than 200 real users. Our analysis shows that the personalized mobile in-app ads are highly correlated with a user's profile. Furthermore, we find that a mobile app was able to predict some personal traits (*e.g.*, gender, age group and parental status) with significantly higher accuracy than random guess. These findings demonstrate the necessity of enforcing strong isolation of third-party content in Android applications to protect both the hosting application and the user data in the embedded application.

1.2.3 Selective Control on Web Tracking

A user's browsing activities across multiple applications can be observed and aggregated by providers of third-party content to infer her or his personal traits and provide targeted advertising and other personalized content. Some third-party tracking scripts do not directly read nor modify the content of the embedding application, which is usually considered harmless. Many first-party applications also track their users' activities. However, such first-party and third-party web tracking practices generally occur without a user's consent and are considered as abuses. To gain trust from end-users, many web trackers (vendors) have provided control mechanisms, *e.g.*, opt-out and DoNotTrack, that allow users to *restrict how they use the collected data*. These server-side mechanisms, however, *do not prevent trackers from collecting data*. Thus, many users seek protection from client-side controls, such as third-party Cookie blocking, ad blockers, and private browsing mode in browsers. Client-side controls usually harm user experience because vendors cannot serve relevant content without collecting users' online footprints.

To achieve a trade-off between privacy and user experience, we propose a new client-side tracking control mechanism, `TrackMeOrNot`, which automatically stops unwanted web tracking when a user visits privacy sensitive content and shares only non-sensitive visits with trackers for a better user experience. `TrackMeOrNot` provides a user with two browsing contexts and enables seamless context switch within a browser tab. `TrackMeOrNot` starts a

navigation with the *anonymous* browsing context and summarizes the web content with its content analysis engine. The user's *persistent* profile is seamlessly switched within the same tab if the content does not violate any privacy policy (preference) defined by the user. Our evaluation with more than 140 user preferences shows that `TrackMeOrNot` achieved high accuracy with low performance overhead in selectively sharing users' online footprints with web trackers.

1.2.4 Monitoring and Understanding Web Content Access Behavior of JavaScript

Third-party JavaScript code is widely embedded in modern web applications. The embedded third-party scripts can help enhance the functionality and improve the experience of a web application. On the other hand, they inherit the full privilege from the websites that embed them. It is generally known that the third-party JavaScript codes can perform lots of privileged and unsafe operations, *e.g.*, `document.write()` and `window.eval()`. It is, however, not well studied that how third-party JavaScript code interacts with the content in the embedding web applications. The lack of knowledge about the behaviors of third-party scripts leads to the under-estimation of the risk of embedding third-party JavaScript code.

To help web developers understand how the embedded JavaScript code accesses their content, *e.g.*, Document Object Model (DOM) elements and Cookies, we design and implement a new monitoring tool, `DOM-Logger`. By inserting access monitoring codes in the implementation of the DOM inside a web browser, `DOM-Logger` can report all the DOM accesses made by any script. Using `DOM-Logger`, we collected and analyzed DOM access logs from Alexa top 100K websites. Our analysis shows that a lot of third-party scripts were abusing their privileges, *e.g.*, accessing Cookies and user inputs and accessing (all) elements created by other scripts. What is worse, those over-privileged scripts, including popular ones provided by Google and Facebook, were commonly found on most websites in our measurement.

1.2.5 Fine-Grained Access Control Policy for the Document Object Model

Embedded third-party JavaScript code has the same privilege of the hosting web application. It can further include scripts from other domains into the hosting application. As a result, a malicious third-party script can compromise the confidentiality and integrity of the hosting application. A web developer can define Content Security Policy (CSP) rules to prevent unknown JavaScript code and other resources from being loaded. However, CSP and other existing web security mechanisms offer only all-or-nothing restriction. Specifically, CSP does not further limit the privilege of permitted scripts, which can be loaded from a compromised web server. Furthermore, such strict and coarse-grained restriction usually harms functionality. To most developers, the benefits of embedding third-party content often outweigh the risks of embedding it. More alarmingly, many developers do not clearly understand what the risks of embedding content from a third party is. Thus, many of them do not enable these security mechanisms, making the protection offered by them in vain.

In order to protect sensitive user data in web applications, it is necessary to limit the privilege of third-party JavaScript code to its minimum. In particular, third-party JavaScript code does not need to access all content (especially the sensitive content) presented in a web application. Existing solutions such as information flow control systems can prohibit sensitive data from being leaked. However, they do not prevent a third-party script from modifying the content in a hosting application.

To help web developers protect both the confidentiality and integrity of their applications, we propose `DOM-ACP`, a fine-grained access control mechanism for the Document Object Model. The mechanism allows web developers to define the access permission (*read* and/or *write*) of any third-party script at per-object granularity through a policy language. The developer defined policies are enforced by a web browser to mediate all JavaScript accesses. We design and implement a prototype of the new mechanism with the Chromium browser that can prevent *direct* read or write access to sensitive content in web applications. We also develop a tool to assist web developers in generating a basic policy for their application.

Our evaluation with popular web applications shows that DOM-ACP is effective in preventing unauthorized JavaScript code from accessing protected DOM objects.

1.3 Dissertation Contributions

In summary, this dissertation makes the following technical contributions to the security research community:

- **New threats:** This dissertation identifies two classes of new threats that compromise the confidentiality and the integrity of a user's indirect data in other applications by a malicious first-party application.
- **New observations:** The findings and observations gained in this dissertation have advanced the community's understanding of emerging threats posed by the practice of embedding third-party content.
- **New techniques:** This dissertation presents two new mitigation techniques to help end users and web developers limit the capability of over-privileged third-party content to prevent unintended behaviors.

CHAPTER 2
POLLUTION ATTACKS AGAINST PERSONALIZED WEB SERVICES AND APPLICATIONS

2.1 Motivation

Personalization is an emerging technology that is embraced by modern web services and applications. By collecting a user's past interactions with a service and then building a user profile based on the collected historical data, a service provider can tailor content to be more relevant to a particular user. Personalization has been widely used by popular web applications in recommending videos (YouTube, Netflix, *etc.*), products (Amazon) and search results (Google) to users. As the primary revenue source of millions of websites, advertisements are also becoming more personalized. The online advertising industry has been putting a concerted effort to increasing the relevance of ads targeted at users by tailoring the ads to their stated or inferred interests.

Serving relevant content not only improves user experience and increases user engagement, but also contributes to growth in revenues. A recent study found that 11.5% of the revenue on a group of e-commerce sites was attributable to personalized product recommendations [1]. On the other hand, a great user experience could attract new users and keep existing users active, thus increases the traffic volume and the ad impressions of a website. Studies have also shown that ads targeted based on a user's online interests have a 40% higher chance in leading to a financial conversion over non-targeted ads [2]. Consequently, the average price online advertisers pay for these targeted ads is 2.6 times higher than non-targeted ads [3].

The increasing use of personalization in modern web applications has also presented a new attack surface for miscreants who try to exploit the new technology for their own interests. In this chapter, we demonstrate that contemporary personalization mechanisms

are vulnerable to exploit. In particular, we show popular web applications are vulnerable to a new class of attack, which we call *pollution attack*, that allows an attacker to alter the customized content they return to users who have visited a web page that is controlled by the attacker. Pollution attack exploits the fact that a service or application employing personalization incorporates a user's past history to customize the content. More importantly, the personalization systems for some web applications do not properly validate the user interaction with them, and their content can be embedded as third parties by attackers. As a result, an attacker can embed other applications as third-party content in the background and hence inject artificial user interactions into personalization systems of these applications.

Based on how a service or application collects user interaction data, pollution attacks can be categorized into two classes. The first class of pollution attacks targets applications that collect data as first parties that a user directly visits. For example, end users interact with websites such as YouTube, Amazon and Google directly. These applications personalize their content based on users' direct activities. The second class targets services that collect data as third parties. In particular, most advertisements are embedded as third-party content by affiliated partner websites a user visits. By logging which websites a user visits, ad networks/exchanges can infer the user's interest and serve ads that target specific user groups. We first demonstrate that attackers can launch the first class of pollution attacks to pollute a victim's profiles on websites that the victim directly visits in §2.2. We then show that a malicious website can pollute the interest profile of a victim on third-party ad networks to display higher-paying (but irrelevant) ads to increase its revenue in §2.3.

The ability to trivially abuse personalization systems to alter a user's content on other web services and applications is especially worrisome because such abuses do not exploit any vulnerability in the user's web browser. Rather, they leverage the design defects in personalization systems of web services and applications. We discuss defense techniques to stop such abuses in §2.4 and related work in §2.5.

2.2 Pollution Attacks against Popular Websites

In this section, we present the first class of pollution attacks that pollute a user's profiles on popular websites that the user directly visits. We first discuss the overview and attack model of the attack in §2.2.1, and then present one concrete attack example against Amazon's product recommendation in §2.2.2. We also apply pollution attack to YouTube's video recommendation and Google's personalized search. Readers interested in these two attacks could find more details in our previous paper [4].

2.2.1 Overview and Attack Model

In this section, we first present a brief overview of personalization as it is used by popular web applications. We then present a model of pollution attacks against websites that end users directly visit.

Personalization

Personalization can help web applications deliver information that is tailored to users' interests and preferences. Users receive more relevant information. Meanwhile, the service providers present content that the user is more likely to purchase or click, which potentially increase the revenue of the service providers.

In addition to the current query (a visit or a search) from a user, the other primary input that personalization algorithms use to customize the content is the user's interaction history. For example, based on the items a user has recently viewed or the keywords a user has searched for, a web application can recommend similar items to the user. Besides the user's past interaction activities, other factors including from geo-location and time of day, may also affect the personalized result. In this section, we focus on how an attacker can pollute a user's interaction history to alter the customized content generated by the personalization algorithm of a web service or application.

Pollution Attacks

The goal of a pollution attack is to affect the personalized content of a website a user directly visits, given a particular query from a user. To achieve such goal, an attacker can trick a user into visiting his website and use techniques such as Cross-site Request Forgery (CSRF) to inject artificial input to a victim's history log on the target website. This attack requires three steps:

1. *Model the personalization algorithm.* We assume that the attacker has some ability to model the personalization algorithm that a website uses to customize its result. In particular, we assume the attacker has the knowledge about what inputs may be used by the personalization algorithm. Such knowledge is often available in published white papers, or can be learned through exploration and experimentation in some cases.

2. *Craft "seed" input to pollute the user's interaction history.* Having some knowledge of what type of input data may be used by a personalization algorithm, the attacker needs to carefully craft seed input that may affect the personalized results. Depending on the application and the query of a user, the seed inputs vary from visits, clicks, queries to any other activity that might be employed by the personalization algorithm. For example, the most recently browsed or searched products may affect the product recommendation results on the homepage of Amazon. The past queries and clicks that are related with a user's current search keywords may affect the personalized search results on Google.

3. *Inject the seed input into the victim's interaction history.* To pollute a user's history, an attacker needs to automatically create interaction event that will be logged by a personalization system. The interaction events have to be associated with the victim's current session. An attacker can directly steal a user's credentials to pollute her/his history, which is very difficult but still possible. A more practical scenario is that

the attacker can use attack vectors such as CSRF to make it appear as though the victim user is interacting with a website (*e.g.*, browsing an item, searching for some keywords).

2.2.2 Pollution Attack against Amazon's Product Recommendation

Amazon's personalization is very direct and obvious to an end user. On the one hand, it makes pollution-based attacks less insidious, as they will be plainly visible to the observant user. On the other, Amazon has the most direct monetization of its screen real estate – users may directly purchase the goods from Amazon – so it is possible that any exploitation of Amazon's personalization can be profitable to an enterprising attacker.

Amazon tailors a customer's homepage based on the previous purchase, browsing and searching behavior of the user. Amazon product recommendations consider each of these three activities individually and explicitly labels its recommendations according to the aspect of the user's history it used to generate them[1]. We focused on the personalized recommendations Amazon generates based on the browsing and searching activities of a customer because manipulating the previous purchase history of a customer may have unintended consequences.

Amazon Recommendations

Amazon displays on a customer's homepage five separate recommendation lists that are ostensibly computed based on the customer's searching and browsing history. Four of these lists are derived from the products that the customer has recently viewed (view-based recommendation lists); the fifth is based on the latest search term the customer entered (search-based recommendation). For each of the view-based recommendation lists, Amazon uses relationships between products that are purchased together to compute the corresponding recommended products. For the recommendation list that is computed based

[1]We performed the study in 2013. Amazon may have changed how it recommends products to users on its homepage today.

on the latest search term of a customer, the recommended products are the top-ranked results for the latest search term. Because customers frequently browse Amazon without being signed in, both the latest viewed products and search term of the customer are associated with the session cookies on the user's browser.

Identifying Seed Products and Terms

Because Amazon computes the view and search-based recommendation lists separately, the seed data required for exploiting each list must also be different.

Visit-based pollution. To promote a targeted product in a view-based recommendation list, an attacker needs to identify a seed product as follows. Given a targeted product that an attacker wishes to promote, the attacker visits the Amazon page of the product and retrieves the related products that are shown on the same page. To test the suitability of these related products, the attacker can visit the Amazon page of that product and then check the Amazon homepage. If the targeted product is shown in a recommendation list, the URL of the candidate related product can be used as a seed to promote the targeted product.

Search-based pollution. To promote a targeted product in a search-based recommendation list, it suffices to identify an appropriate search term. If automation is desired, an attacker could use a natural language toolkit to automatically extract a candidate keyword set from the targeted product's name. Any combination of these keywords that successfully isolates the targeted product can be used as the seed search term for promoting the targeted product. For example, to promote product "Breville BJE200XL Compact Juice Fountain 700-Watt Juice Extractor", an attacker can inject search term "Breville BJE200XL" to replace an Amazon customer's latest search term through CSRF.

Injecting Views/Searches

The attacker embeds the Amazon URLs of the desired seed items or search queries into a website that the victim's browser is induced to visit with CSRF. For example, if one

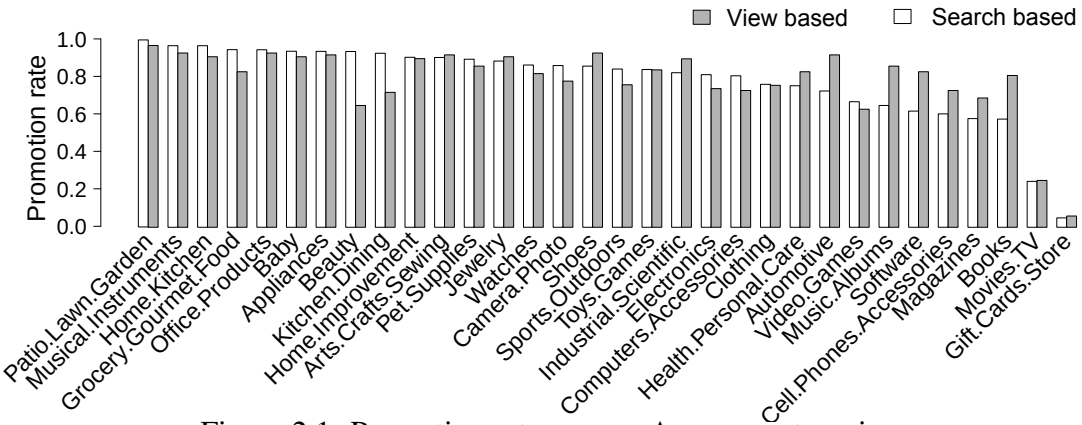


Figure 2.1: Promotion rates across Amazon categories.

seed search term is "Coffee Maker", the seed URL would be something like http://www.amazon.com/s/?field-keywords=Coffee+Maker. Similarly, an attacker could embed the URL of a seed product into an invisible *img* tag as the *src* of the image. When a victim visits the attacker's website, Amazon receives the request for that particular query or item and customizes the content of victim's Amazon homepage based on that interaction event.

Experimental Design

To evaluate the effectiveness of the pollution attack against Amazon, we conducted two experiments. The first experiment measured the effectiveness of our attack when targeted toward popular items across different categories of Amazon products. The second quantified the effectiveness of our attack on randomly selected, mostly unpopular Amazon products.

Popular Products. Amazon categorizes sellers' products into 32 root categories. To select products from each category, we scraped the top 100 best-selling products in each category in January 2013, and launched a separate attack targeting each of these 3,200 items.

Random Products. To evaluate the effectiveness of the pollution attack for promoting arbitrary products, we also selected products randomly. We downloaded a list of Amazon Standard Identification Number (ASIN) [5] that includes 75,115,473 ASIN records. Since each ASIN represents an Amazon product, we randomly sampled ASINs from the list and constructed a set of 3,000 products that were available for sale. For every randomly selected product in the list, we recorded the sale ranking of that product in its corresponding category.

Evaluation

Because Amazon computes search-based and visit-based recommendations based entirely upon the most recent history, we can evaluate the effectiveness of the pollution attack without using Amazon accounts from real users. Thus, we measured the effectiveness of our attack by studying the success rate of promoting our targeted products for fresh Amazon accounts.

Promoting Products in Different Categories. To evaluate the effectiveness of the pollution attack for each targeted product, we checked whether the ASIN of the targeted product matches the ASIN of an item in the recommendation lists on the user's customized Amazon homepage.

Figure 2.1 illustrates the promotion rate of target products in each category. The view-based and search-based attacks produced similar promotion rates across all categories, about 78% on average. Two categories had significantly lower promotion rates: Gift-Cards-Store and Movies-TV (5% and 25%, respectively).

To understand why these categories yielded lower promotion rates, we analyzed the top 100 best selling products for each category. For Gift-Cards-Store, we found that there are two factors that distinguish gift cards from other product types. First, the gift cards all had similar names; therefore, using the keywords derived from the product name results in only a small number of specific gift cards being recommended. Second, we found that searching any combination of keywords extracted from the product names always caused a promotion of Amazon's own gift cards, which may imply that it is more difficult to promote product types that Amazon competes with directly.

Further investigation into the Movies-TV category revealed that Amazon recommends TV episodes differently. In our attempts to promote specific TV episodes, we found that Amazon recommends instead the first or latest episode of the corresponding TV series or the entire series. Because we declared a promotion successful only if the exact ASIN appears in the recommendation lists, these alternative recommendations are considered failures. These cases can also be considered successful because the attack caused the promotion of very

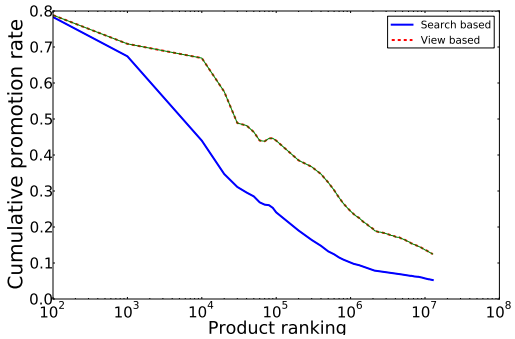


Figure 2.2: Cumulative promotion rates across varying product ranks for different Amazon pollution attacks.

similar products. Therefore, we believe that for all categories except for Gift-Cards-Store, an attacker has a significant chance of successfully promoting best-selling products.

Promoting Randomly Selected Products. We launched pollution attacks to promote 3,000 randomly selected products. We calculated the *Cumulative Success Rate* of products with respect to their rankings. The Cumulative Success Rate for a given range of product rankings is defined as the ratio of the number of successfully promoted products to the number of target products in that range.

Figure 2.2 shows the cumulative promotion rates for different product rankings for the two different types of pollution attacks. As the target product decrease in popularity (*i.e.*, have a higher ranking position within its category) pollution attacks become less effective. This phenomenon reflects a limitation of Amazon recommendation algorithms, not our attack. Products with low rankings might not be purchased as often; as a result, they may have few and weak co-visit and co-purchase relationships with other products. Our preliminary investigation finds that products which rank 2,000 or higher within their category have at least a 50% chance of being promoted by a visit-based pollution attack, and products with rankings 10,000 and higher have at least a 30% chance to be promoted using search-based attacks.

2.3 Pollution Attack against Targeted Advertising

In this section, we present the second class of pollution attacks that pollute a user's profiles on services and applications that the user usually not directly visits, *i.e.*, third-party applications. In particular, we demonstrate this class of pollution attacks using targeted online advertising as an example. We present a new ad fraud mechanism that enables publishers to increase their ad revenue by exploiting the role played by the user's online interest profile in the ad selection process. Our attack exploits the fact that advertisers mainly set up campaigns to target users with specific online interests and are willing to pay higher for such users. Since the user's interest profile is inferred by third-party advertising and tracking libraries based on the web pages a user visits, it is vulnerable to exploits that use Cross-Site Request Forgery (CSRF) [6], clickjacking [7] or cross-site scripting (XSS) [8] that can pollute users' profiles by generating camouflaged requests to web pages not explicitly visited by them. A fraudulent publisher can use these exploits to pollute the profiles of users visiting the publisher's website to mislead advertisers and the ad exchange to deliver more lucrative ads to these users, and thereby increase the publisher's ad revenue.

While the above described attack seems intuitive, it is not trivial to design and launch the attack such that it is practical, effective, and lucrative. To the best of our knowledge, we are the first to design and successfully deploy a pollution attack on the existing targeted advertising ecosystem. Achieving this requires addressing the following challenges which also form the main contributions of our work. First, the attack should not require any explicit cooperation from the ad exchange or advertisers, and should be effective for the two commonly used ad targeting mechanisms – behavioral targeting and re-marketing. Second, polluting user profiles should be effective even without explicit knowledge about external factors that impact ad revenue (campaign budgets, bid costs, publisher preferences and ad inventory, *etc.*). Third, it should be feasible to load the pollution content in a camouflaged manner such that it is not discernible by the users while deceiving the ad exchange and

advertisers. Finally, the polluted user profile should result in biasing the ads targeted at the user towards the intended higher-paying advertisers.

To address the above described challenges, we set up and validate the attack against one of the largest ad exchanges, DoubleClick, and study the monetary value of the attack for live publisher web pages. Instead of polluting live traffic, we emulate user traffic to the publisher websites by replaying web traces collected from 619 real users from 264 distinct IP addresses and recording all ads delivered to these emulated users. This setup enables an end-to-end characterization of the different aspects of the attack under controlled settings that is otherwise not feasible. Our results show that the attack is *successful* and *effective* in deceiving DoubleClick to deliver higher-paying ads on the fraudulent publisher's website. Using our attack, the polluter can influence up to 74% and 12% of the total ad impressions for re-marketing and behavioral pollution, respectively. Finally, we show that the attack is *lucrative*, enabling the fraudulent publishers to increase their ad revenue on average by 33%.

The main contributions of this section are:

- To the best of our knowledge, we are the first to demonstrates a practical application of a profile pollution attack with clear monetary benefits to the attacker. We perform an end-to-end validation and characterization of the attack on the online advertising ecosystem and study the monetary value to publishers.

- We present novel approaches for selecting content used for polluting users' profiles in a way that influences the two most commonly used ad targeting mechanisms – re-marketing and behavioral targeting – while not being discernible by the polluted users.

- We validate our attack against one of the largest ad exchanges, DoubleClick, and show that it is *successful* and *effective* in deceiving DoubleClick to deliver higher-paying ads on the fraudulent publisher's website. Using our attack, the polluter can influence up to 74% and 12% of the total ad impressions for re-marketing and behavioral pollution,

respectively.

- Finally, we show that the attack is *lucrative*, enabling the fraudulent publishers to increase their ad revenue by 4% to 23%. We also present a detailed study of the primary factors that influence revenue generated and discuss potential countermeasures.

2.3.1 Ad Targeting and User Profiles

In this section we describe the ad targeting mechanisms available to advertisers [9] and discuss the critical role played by a user's online interest profile in the existing ad ecosystem.

Ad Targeting Mechanisms

Contextual Targeting. Contextual targeting involves matching the ad with the context of the page that it is displayed on (and ignores the visitor interest profile). The targeting is implicit and the user's online interests are largely ignored: a car insurance company will place ads on auto-related sites because it is assumed that visitors to the site are likely to own a car (or want to) and will need insurance.

Re-Marketing. Re-marketing is used by advertisers to target users who, in the past, have indicated a very specific interest in a particular product. For example, consider a user who visits a car insurance website, clicks on a link to get a quote, but leaves without buying the insurance offered. The insurance company (via the ad exchange) can then target this user with re-marketing ads, *e.g.*, showing insurance discounts. These ads will be delivered to the user on other websites, which may be completely unrelated to cars or insurance, to lure the user back to finish the purchase. Here, the advertiser targets a user by exploiting a very specific signal.

Behavioral Targeting. Behavioral targeting is used by advertisers that target users who have shown an interest in some categories (*e.g.*, cars or college football). This mechanism goes beyond the "single domain" aspect of re-marketing, and selects ads that might relate to the user's online interests as observed from her browsing patterns. This form of targeting

often results in ads that may be unrelated with the page being viewed [10]. For example, with behavioral targeting, a user might be targeted with car insurance related ads (potentially from a company she did not visit online) on a website about Food & Nutrition simply because she visited multiple different car insurance related websites, and the ad exchange profiled her to be interested in car insurance.

User Profiles and Targeted Ads

Behavioral targeting and re-marketing make explicit use of the user's online interests that are profiled by the ad exchange and other third party trackers. This is achieved by installing third party JavaScript tracking code provided by the ad exchange on websites that users' browse. The tracking code extracts details about the page (*e.g.*, exact URL, `meta` tags about keywords, description, *etc.* [11]) and transmits this along with the user's cookie identifier. This information, along with other information that the ad exchange has about the website, is used to profile the user's interests and are offered to advertisers as targeting options. A user's interest profile for behavioral targeting is represented as a set of semantic categories, structured as a hierarchy (*e.g.*, Movies→Action Films→Superhero films). For re-marketing, the ad exchange simply maintains a list of users (cookie IDs) that visit a specific page on the advertiser's website.

As is evident, the user's interest profile forms an integral component of the ad selection process. Advertisers assign a monetary value using cost-per-click (CPC) or cost-per-mille (CPM) directly to the user's online interests and are willing to pay up to 2.6 times higher to target ads at users with a desired profile [3]. Moreover, as we show in Section 2.3.4, although a user may have online interests accumulated over a long time period, short term browsing activity can significantly impact the user's profile and consequently change the type of ads that a user receives. Our attack exploits this critical aspect and enables publishers to pollute user profiles towards ad categories that generate higher revenue. In the following section we provide an overview of the attack and present techniques for profile pollution

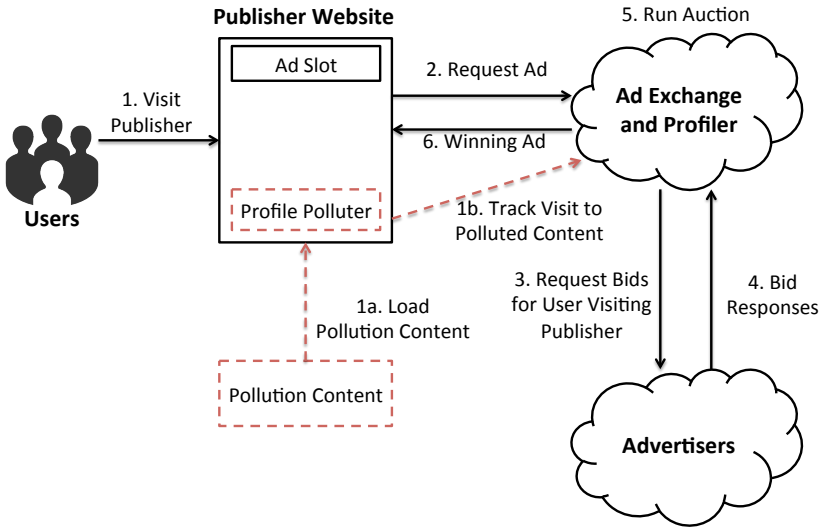


Figure 2.3: An overview of the profile pollution attack

that are specific to the ad ecosystem and the commonly used ad targeting mechanisms.

2.3.2 Profile Pollution Attack

The profile pollution attack (which we also refer to as a fraud mechanism) introduces a new entity in the ad ecosystem that we call *profile polluter*. Figure 2.3 shows the interaction of the profile polluter with the rest of the ad ecosystem (dashed lines). Specifically, the primary steps involved in a successful attack are:

1. The profile polluter identifies and downloads content in order to pollute user profiles.
2. A user visits the polluter page (which can be hosted at the publisher's website) and pollution content is loaded first in a camouflaged manner. (steps 1 and 1a in the figure).
3. This signals the ad exchange of a legitimate browsing event by the user, and the user's profile is impacted (step 1b).
4. When the user navigates to another page on the publisher's website, the ad exchange is deceived in using this modified profile in soliciting bids for ads (steps 2-5).
5. The publisher's revenue increases if the winning ad is from an advertiser that bids higher

to target the polluted user (step 6).

The attack focuses on polluting users to influence behavioral targeting and re-marketing ad campaigns, as they explicitly make use of the user's online interest profile. In order to simplify the description, we assume that the publisher also plays the role of the profile polluter. The readers should note under this assumption the attack could only impact ads of user's next visit, as the website content and pollution content are loaded by browser in parallel in each visit.

Identifying Pollution Content

Pollution Content for Re-Marketing. A re-marketing campaign is set up by integrating a few lines of JavaScript code, *i.e.*, the *re-marketing script*, which is provided by the ad exchange, in the advertiser's website. The JavaScript code encodes the unique identifier of the advertiser and the associated re-marketing campaign. When a user visits the re-marketing enabled advertiser website, these identifiers along with the user's ad exchange cookie are transmitted to the ad exchange. This enables the ad exchange to tag the user and track her interactions on the advertiser's website. The tagged user is then easily re-identified later on other websites and is targeted with ads from the advertiser. Consequently, a user's past browsing history and online interests do not impact re-marketing ads.

This script can be easily detected by parsing the HTML code of a web page[2]. Thus, a simple approach to find content for re-marketing pollution is to parse web pages of advertisers belonging to high-paying categories and identify those that host re-marketing scripts.

Pollution Content for Behavioral Targeting. The approach of simply scanning websites in a directory service is not sufficient for finding content for behavioral pollution as the ad exchange categories may not match those of the directory service. Alternatively, the polluter can exploit the ad preference dashboards made available by large ad exchanges to

[2]DoubleClick itself provides instructions on how their Tag Assistant detects re-marketing scripts. For more details, see https://support.google.com/tagassistant/answer/2954407?hl=en

build an offline map between web pages and the category label assigned to these web pages. Specifically, the polluter can impersonate a user with a blank profile (delete all cookies and create a fresh browser profile), browse pages from a specific category and record the corresponding profile generated by the ad exchange. This map can then be used to select pollution content. Unlike re-marketing based pollution, the impact of behavioral pollution on altering user profiles towards more lucrative advertisers depends on the users' existing online interest profile. We empirically evaluate this impact across diverse user profiles in Section 2.3.4.

Hosting and Loading Pollution Content

The pollution content hosted by the fraudulent publisher should be loaded by the user's browser in a way that is not discernible by the user and ad exchange. While there are many ways to fabricate such camouflaged requests, such as `CSRF`, `XSS` and `Clickjacking` *etc.*, in this chapter we assume the pollution content is loaded using cross reference issued by hidden HTML `iframes`. These `iframes` are located outside the viewing area of the browser or layered underneath other content, and are used to reference and load pollution content. The loading of such content takes place in the background and is completely hidden from the user. Moreover, since approaches for frame-busting are not ubiquitously deployed, simple approaches can be used to hide the frame content from web crawlers.

Attack Monetization – CPM and CPC

An important property of the attack is that it can be used to further boost the revenue generated by existing click and impression fraud mechanisms. This can be achieved if the bot master has control over the user's browser such that it can pollute user profiles to maximize the impact of the fraud. When deployed in isolation of existing fraud mechanisms, the attack is most effective for CPM-based ad campaigns. This is because CPM-based campaigns, which are the most common campaigns for display ads [11, 12], provide

consistent cash flow to the publisher, regardless of whether visitors click on the potentially unrelated ads. In the rest of the chapter we focus on CPM-based campaigns and assume that the attack is deployed as a standalone attack without deploying additional fraud methods.

2.3.3 Attack Setup

We set up the attack as follows. Instead of driving live traffic, we emulate users browsing the websites with web traces. A few domains from the users' traces are selected as the fraudulent publishers. As we do not have control over these websites, the profile polluter is separated from the fraudulent publisher and is responsible for polluting the emulated user traffic immediately after loading the publisher's page to approximate a publisher that pollutes his own users. A distributed testbed of 200 nodes spread across the world (using PlanetLab) is used to generate web traffic to ensure location diversity. Since the traffic is emulated from browsers that we control, an ad crawler is used to record all the DoubleClick ads delivered to the emulated users. The recorded ads are analyzed and the revenue is estimated using publicly available CPM index values published by DoubleClick [13]. We also set up our own website as a fraudulent publisher to characterize the effectiveness of the attack.

User Web Traces & Profiles

Our attack setup replays *complete* web traces from real users to characterize and validate the attack. This is important because the ad revenue is not only impacted by the frequency with which users visit the publisher page but also depends on the user's online interest profile before and after pollution; the pollution impact depends on websites visited *prior* to pollution and the duration of the impact depends on websites visited *after* pollution.

We use web traces of real users from a Chrome extension installed by more than 700 users who have been using the extension for 2 years for research purpose. The functionality of the extension was modified to record all the web-page URLs visited by the user for a one

week period (March 10th, 2014 - March 16th, 2014)[3]. In this time period, we collected a total of 224,855 page visits from 619 unique active users. Our dataset is diverse and consists of users using the extension across the world.

We create multiple copies of each user's profile to load under different pollution settings. The user web traces are replayed to generate profiles that are polluted by forwarding the request to the profile polluter after visiting the fraudulent publisher. We also create clean profiles by replaying user web traces bypassing the profile polluter. To eliminate the impact from time and location on distribution of ads, each user's profiles are generated by replaying her web trace at the same time from the same IP address. This provides a seamless approach to measure the extent to which the pollution impacts the type of ads targeted at the user. Thus, for every experiment presented in the following two sections, we record the ads targeted at the user with and without profile pollution.

Pollution Content

As described in Section 2.3.2, user profiles are polluted in order to mislead the ad exchange and advertisers from more lucrative verticals to target ads at users visiting the fraudulent publisher's web page. We pollute each user by generating camouflaged visits to three websites from the top three most expensive display ad categories of *Health*, *Business* and *Education*. Beyond the top three ad categories, we pick two additional categories of *Sports* and *Shopping* to study the attack on less valuable ad categories.

Polluting for Behavioral Targeting. In order to find websites that alter the user's interest profile towards the above mentioned categories, we first filter websites from the corresponding Alexa category that contain the DoubleClick tracking script. For each website in this list, we use the Google Ad Preferences Dashboard [14] to build a map between the websites and categories that are consistent with DoubleClick. Table 2.1 provides the three websites selected for each category.

[3]IRB approval was granted and users were notified about the type of data collected and the intent of use for research purposes

Table 2.1: The websites we use for polluting users' profiles in the five ad categories.

Google Category	Alexa Category	Re-marketing Pollution Contents	Behavioral Pollution Contents
Health	Health	*eyemagic.net*	*intensemuscle.com* *bimabazaar.com* *allacqueredup.com*
Business	Business	*incorporate.com*	*bloomberg.com/news/insurance/* *bloomberg.com/news/finance/* *bloomberg.com/news/industries*
Educaton	Reference	*asuonline.asu.edu*	*universando.com* *campusleader.com* *graphs.net*
Shopping	Shopping	*teleflora.com*	*alterationsneeded.com* *modernsalon.com* *viloux.com*
Sports	Sports	*moenormangolf.com*	*bloguin.com* *retospadel.com* *golftechnic.com*

Polluting for Re-Marketing Targeting. Similar to the previous approach, we filter websites in the Alexa category that host the re-marketing script from DoubleClick. In addition to verifying that the category matches, we also verify that the re-marketing campaign is active. Table 2.1 lists the websites used for re-marketing pollution for each category.

Publisher Web Page

The complete attack is validated on two different type of publisher web pages.

Live Websites. We validate the attack on existing live publishers whose ad revenue is impacted by the dynamic content hosted by them as well as pre-existing preferences about type of ads that are allowed to be targeted. To this end, we select the top 19 most visited websites that host DoubleClick ads from the user web traces. Instead of compromising these websites to host pollution content, we set up the profile polluter as a separate entity. When emulating traffic traces, we forward the user to the profile polluter immediately after visiting any one of these 19 websites. We use results from these publishers primarily to estimate the revenue generated by the attack (Section 2.3.5).

Controlled Publisher. In order to form a baseline of the effectiveness of the attack, we set up our own publisher website and sign up with AdSense [9]. The publisher website has two display ad slots (top banner display ad and a side display ad) and uses the default settings

provided by AdSense. Since AdSense requires the website to host some content before approving it, we upload static content that describes the different ad targeting mechanisms available to advertisers. Visiting the web page with a blank profile results in DoubleClick profiling the user with interests belonging to the *Computers* category. Similar to the above setup, the profile polluter is separated from the controlled publisher.

Trace Emulator and Ad Crawler

A critical component of the attack setup is a distributed infrastructure to emulate web traffic by replaying the traces and recording all the ads delivered to the emulated user.

Trace Emulator. We develop a distributed infrastructure based on the PlanetLab testbed that is able to emulate real user web traffic. The trace emulator consists of a central control server and 264 worker nodes distributed across the world. The server maintains a list of tasks that are fetched by distributed workers periodically. Given a task containing the URL to visit and a unique user ID, the worker node instance loads one profile of the corresponding user, visits the assigned URL, records all the ads displayed on the web page and the associated metadata, and sends this information back to the central server. The user's profile is updated accordingly after visiting the assigned URL.

Ad Crawler. Collecting measurements about display ads requires the ability to disassemble the elements of a web page, identify ad elements and associate these with particular categories. Existing ad monitoring and blacklisting tools – AdBlock [15] and Ghostery [16] – work by matching URL patterns embedded in a web page against a set of blacklist patterns, and cannot look deeper into the element and reason about it. The task is made even more difficult by complex DOM structures, deep nesting of elements, and dynamic JavaScript execution, that is found on a large fraction of pages on the Internet today. To address these challenges we extend the PhantomJS headless browser[4] to reliably extract the ad elements of a page, identify the actual landing pages for the ad elements, and associate the ads with

[4]http://phantomjs.org

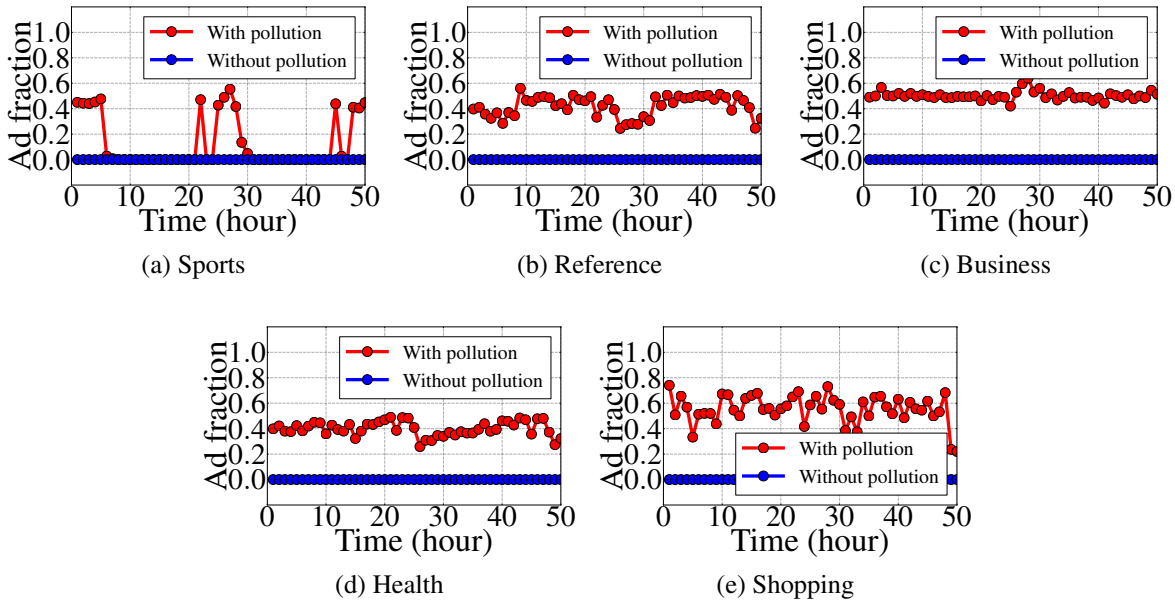
specific semantic categories. The current implementation of the ad crawler is limited to ads delivered by DoubleClick.

Ad Revenue Estimation

The final component of our setup is to estimate the ad revenue the fraudulent publisher generates. We estimate the revenue by using the publicly available report provided by Google that ranks the CPM cost of different ad verticals (categories) and associates with each vertical a relative cost index [13]. In the Google report, the index for the three most expensive categories of `Health`, `Business` and `Job & Education` were 257, 221 and 200 correspondingly. The least expensive category was `Law & Government` with an index of 46. We further manually mapped each of the 13 top-level Alexa categories to one of the ad verticals used in the published Google report. Our revenue estimation analysis always compares the revenue generated by the pollution attack with the baseline revenue computed by running the exact same experiment without pollution.

2.3.4 Validation and Effectiveness of the Attack

In this section we evaluate the extent to which profile pollution impacts the ads by deploying the attack on the controlled publisher web page. The primary metric used for the evaluation is the relative change in ads from the desired ad category (behavioral pollution) or domain (re-marketing pollution) with and without pollution. For both user profile sets, the trace emulator first visits every website in a user's trace to ensure that all users have an online interest profile. We then take one set of user profiles and pollute all users only once. Subsequently, users from both sets visit the controlled publisher page once every hour for a duration of 50 hours. For each visit to the publisher web page the ad crawler captures all the ads.



(a) Sports (b) Reference (c) Business

(d) Health (e) Shopping

Figure 2.4: Effectiveness of pollution attacks against re-marketing ad campaigns across different ad categories.

Pollution Using Re-Marketing Campaigns

Figure 2.4 shows, for each pollution category, the fraction of ads received with and without profile pollution across all users. As expected, we observe that only the polluted users receive ads from the category used in the pollution. Surprisingly, we observe that across all categories, re-marketing ads aggressively target users, both in terms of time between the pollution and first ad shown, and number of ads: across *all* pollution categories users receive ads from the intended advertisers immediately in the very first visit to the publisher's webpage, and approximately 40–50% of all display ads are from these advertisers. We also verified the distribution of ads across users (not shown) and found that *all* the users received ads from the re-marketing campaigns used for pollution.

While the pollution is highly effective once it is triggered, advertisers may set up specific rules to trigger the campaign that can impact the publisher's ad revenue. First, as seen in Figure 2.4 advertisers can set up time based triggers. For example, the advertiser `moenormangolf.com` from the Sports category set up the campaign to only run during 8 hours of the day, causing a diurnal pattern in the targeted ads. Alternately, campaigns

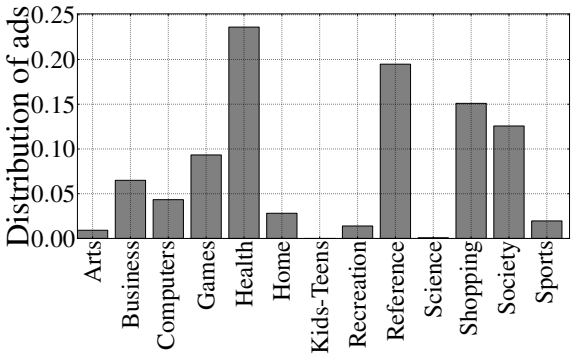
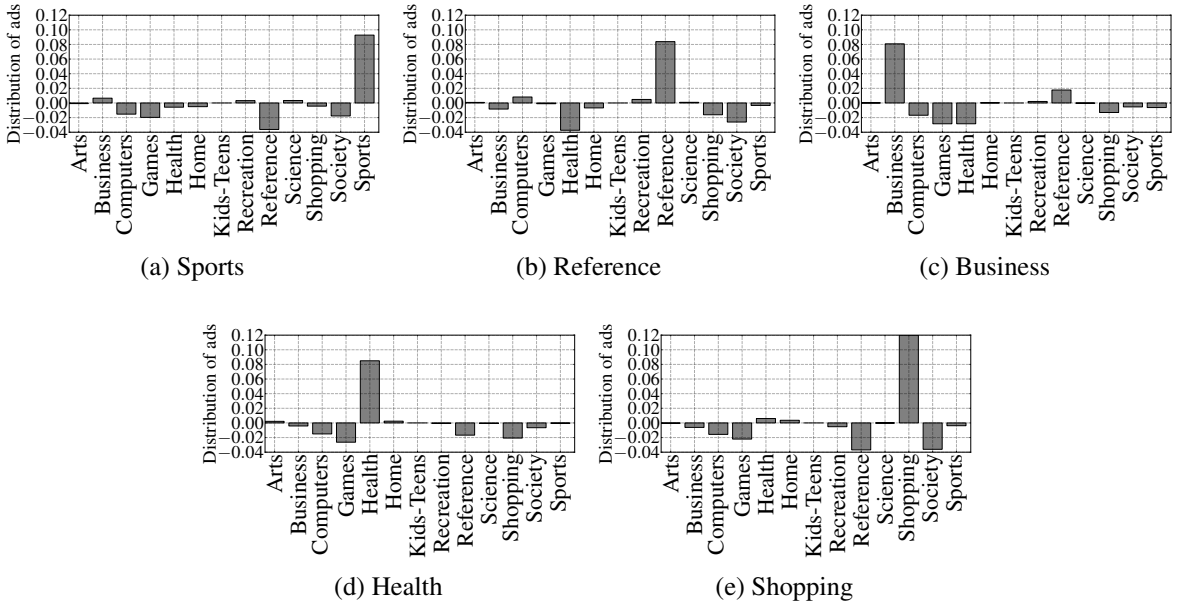


Figure 2.5: Distribution of ads across the 13 top-level Alexa categories.



(a) Sports (b) Reference (c) Business

(d) Health (e) Shopping

Figure 2.6: Change in the distribution of ads.

can be set up with frequency caps or they may be paused by the advertiser. Additionally, the advertiser may set up the campaign with a more complex control flow of user actions (*e.g.*, went to homepage, placed things in the cart, but never checked-out) and trigger the campaign only when a user completes all the steps. Thus, the primary challenge in effectively exploiting re-marketing campaigns is to select pollution content that accounts for such specific trigger rules.

Pollution Using Behavioral Targeting Campaigns

Impact on Ad Categories. Figure 2.5 shows the distribution of ads across the 13 top-level Alexa categories for user profiles that were not polluted. We observe that the ad

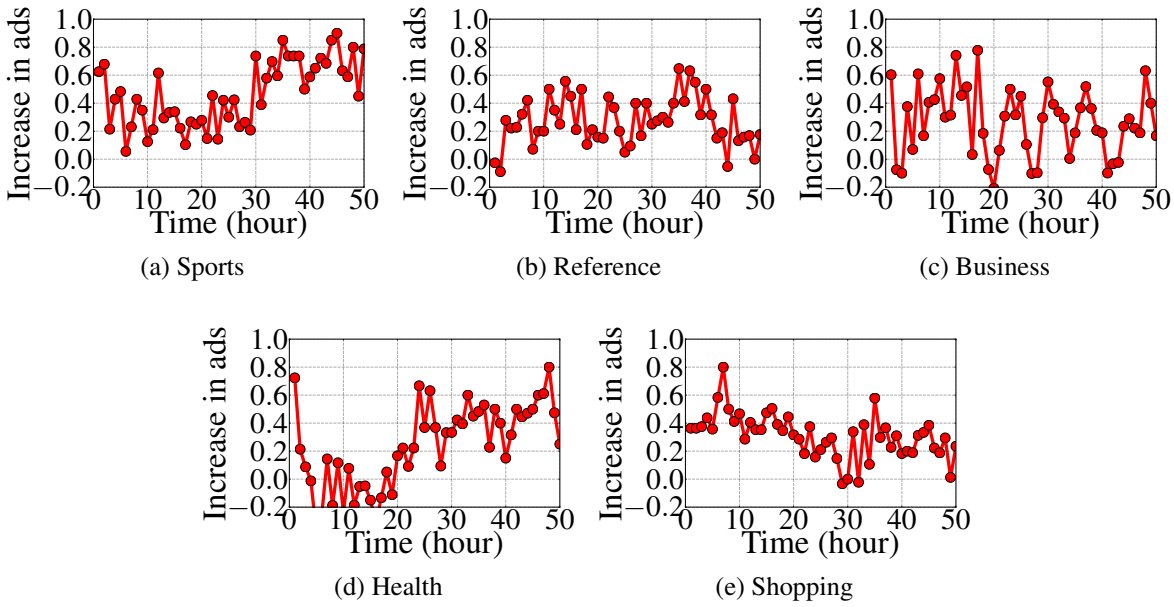


Figure 2.7: Percentage increase in ads (pollution - no pollution) from the polluted category.

distribution spans multiple categories as users have diverse online interests. We use this baseline distribution and compute the relative change in the distribution of ad categories after pollution. Figure 2.6 shows the relative change in the ad categories across users which validates the effectiveness of our pollution attack – there is a clear increase in ads in the polluted category with a maximum increase of 12% for the *Shopping* category. Moreover, the pollution manages to increase the number of ads in categories that a user already received prior to pollution. For example, the fraction of ads in the *Health* category increases from 23% to 31%.

Temporal Impact. Finally, we study the temporal effect of the pollution. Figure 2.7 shows the relative increase in fraction of ads received from the category used for pollution. We observe that the effect of the pollution is immediate and leads to an increase in ads from the desired category. Moreover, the effect of the pollution persists over the entire time duration of the experiment. This indicates that categories introduced artificially as an effect of the pollution have a lasting influence on the ads received by the user.

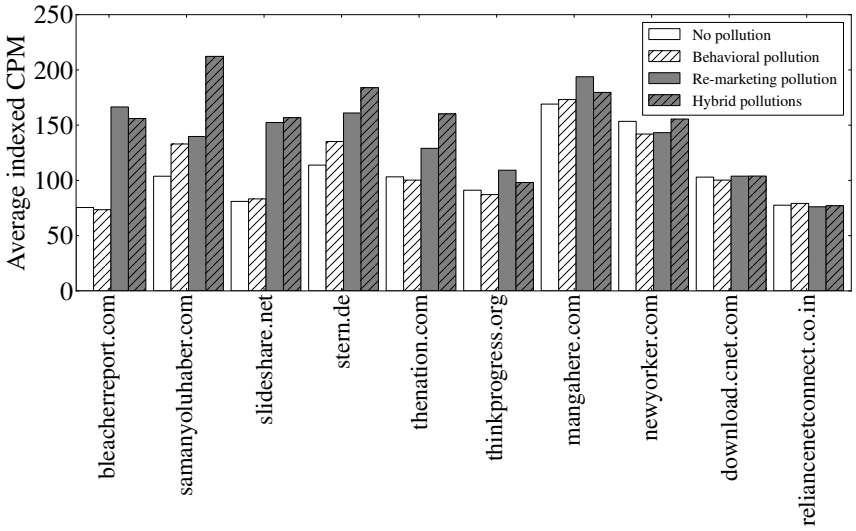


Figure 2.8: Average indexed CPM across the top 5 and bottom 5 selected websites before and after pollution.

Table 2.2: Details of revenue experiments, showing the top 5 and bottom 5 websites we designated as fraudulent publishers ranked by relative change in indexed CPM using profile pollution.

Site	Alexa global rank	Avg page views per user per day	Num users	Avg page views per day	Change (% behavioral)	Change (% re-marketing)	Change (% hybrid)
bleacherreport.com	231	3.96	133	527	-2.60	120.64	106.81
samanyoluhaber.com	1,396	13.44	85	1142	28.11	34.67	104.52
slideshare.net	120	2.29	146	335	2.78	88.15	93.57
stern.de	1,691	2.58	60	155	18.75	41.43	61.54
thenation.com	13,835	1.50	88	132	-2.88	24.99	55.15
thinkprogress.org	3,960	1.37	91	125	-4.37	19.90	7.70
mangahere.com	1,903	72.71	52	3781	2.43	14.63	6.25
newyorker.com	2,432	1.93	159	307	-7.49	-6.67	1.37
download.cnet.com	104	4.48	69	309	-2.62	0.86	0.95
reliancenetconnect.co.in	1,694	1.79	102	183	2.07	-1.85	-0.61

2.3.5 Revenue Estimation for Live Publishers

In this section, we deploy the attack on live publisher websites and estimate the revenue generated by the attack for these publishers. Unlike the controlled publisher setting, there are a number of factors like the hosted content, popularity of the website, and ad preferences setup by the publisher that impact the ad revenue. While it is not feasible to explain the specific factors that impacts the publisher's revenue, we seek to empirically measure the overall impact of the pollution on the revenue of live publishers.

As described in Section 2.3.3, we select the top 19 most frequently visited websites from the web traces that host DoubleClick ads as the "fraudulent" publishers. When replaying the

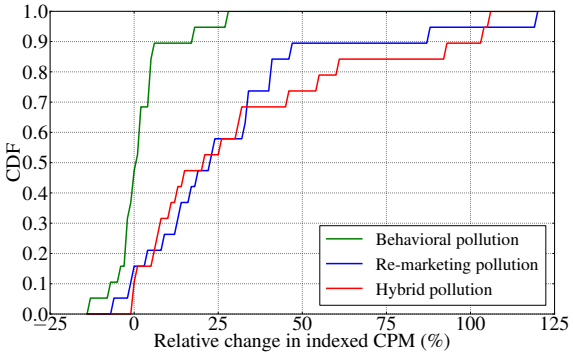


Figure 2.9: Distribution of the relative increase in the indexed CPM across the 19 selected websites.

web traces, every visit to one of these 19 domains is followed by visiting the profile polluter. We emulate these traces four times in parallel for the following four pollution configurations - without pollution, behavioral pollution, re-marketing pollution, and hybrid (both) pollution using the pollution content shown in Table 2.1. The revenue is estimated using the CPM index [13] data reported by DoubleClick.

Aggregate CPM Index Change

Figure 2.9 shows the relative change in the CPM index for the three pollution configurations across the 19 websites. Overall, we find that behavioral pollution is not as effective as re-marketing based pollution; for almost 80% of the websites the change in the indexed CPM is not significant ($\pm$ 5%). On the other hand, re-marketing based pollution does significantly and consistently increase the relative indexed CPM; an increase of 4–120% for about 80% of the domains.

To better understand these distributions, Table 2.2 provides the traffic statistics along with the relative change of CPM index for the top five and bottom five performing domains ordered by the CPM index with hybrid pollution. Figure 2.8 shows the average indexed CPM for the same 10 websites. We make a number of observations from this data:

Website Ranking and Traffic Patterns. Across the five best and worst performing websites we do not observe any correlation between the website ranking or traffic patterns with the revenue generated by either one of the three pollution configurations. This indicates

that our attack is able to deceive the ad exchange in targeting high value ads even on websites that are ranked much lower or have highly varying traffic patterns.

Varying Performance of Behavioral Pollution. We observe that behavioral pollution does not consistently increase the ad revenue for the fraudulent publisher. Among the top five websites listed in Table 2.2, *bleacherreport.com*, *slideshare.net* and *thenation.com* yield a negative or very low increase in the average CPM index. Looking into the logs, we find that the behavioral pollution of the emulated traffic to these websites was ineffective. For example, 83% and 85% of the ads targeted on *bleacherreport.com* were from a single advertiser, *ford.com*, before and after behavioral pollution, respectively. Similarly, 100% and 93% of the ads on *slideshare.net* were from *academy.com* before and after behavioral pollution, respectively. On the other hand, re-marketing and hybrid pollution for these domains was effective and led to a significant increase in ad revenue. This potentially indicates that these websites have pre-sold their ad inventory and consequently behavioral pollution was ineffective. However, re-marketing based pollution manages to override this pre-sold ad inventory, potentially because of the higher CPM and CPC costs associated with these ads.

Low Yield Re-marketing Pollution. As discussed in Section 2.3.4, re-marketing based pollution leads to aggressive targeting of users independent of their online profile. However, we observe that for *newyorker.com*, *download.cnet.com*, and *reliancenetconnect.co.in* all three pollution configurations are ineffective. None of the three domains received ads from the advertisers used for re-marketing pollution even when users visiting other domains were targeted with the re-marketing ads. Moreover, the behavioral pollution was also ineffective for these domains. For example, on *reliancenetconnect.co.in*, between 65%-73% of the ads targeted at users before and after pollution (all three pollution types) were automobile related ads from domains like *mazdausa.com*, *avis.com*, *budget.com* and *driveamazda.com*. This potentially indicates a scenario where the publisher website is explicitly configured to only receive automobile related ads making the different pollution mechanisms ineffective.

2.4 Countermeasures

In this section we discuss countermeasures and best practices that different entities can adopt in order to mitigate or at least minimize the attack surface.

General Countermeasures. Commonly, websites are not supposed to be framed within another website as part of an `iframe` [17]. Therefore, using `X-Frame-Option` or deploying a "frame-busting" method can make it more difficult for the polluter to abuse innocent websites for the purpose of pollution fraud (other methods, such as pop-unders can still be used, but are easier to detect).

For a website that allow other websites to embed it, it should check the browsing context when a user interacts with it. The user's interaction should not be logged if its web page is not loaded in the main frame. Similarly, ad and tracking libraries should check the user's browsing context before logging any data. In addition, web services and applications should implement defenses against CSRF attacks for critical endpoints that may change a user's history. One effective defense is associating an user interaction event with a server generated unforgeable token. Thus, attackers can not inject artificial inputs any more because they can not obtain a valid token.

Advertisers. Advertisers should protect their ad campaigns against pollution attacks by targeting audiences that have very specific interests. This effectively raises the bar for the polluter to find relevant pollution content impacting a large number of users. For example, finding the appropriate pollution content for the category *Jobs & Education→Education→Distance Learning* may be more difficult to compared to finding pollution content for *Education*. Similarly, a re-marketing campaign that targets users with a specific flow in the website, *e.g.*, users who logged in, placed an item in a cart but did not check out, is more difficult to compromise compared to targeting all users who visited the web page of the advertiser. We note that the downside of such fine-grained audience targeting is that it may reduce the size of the target audience.

Ad Exchange and Ad Networks. Recent work, like ViceROI [18], aims to detect click

spam by comparing the revenue per user for a fraudulent publisher with a baseline set of ethical publishers. While this approach is limited to catching click spam, ad networks and ad exchanges should deploy similar approaches to detect impression fraud caused by anomalous revenue changes in the fraudulent publisher's ad revenue. Even though the attacker has control over his ad revenue through configuring the attack settings (e.g. pollution content, ad preference, and amount of polluted users, etc.), the deployment of systems like ViceROI could reduce the ad revenue generated from profile pollution.

Ad exchanges like DoubleClick do not check for the domain in which the re-marketing script is being executed. Consequently, it is sufficient for the polluter to simply copy the JavaScript provided by the ad exchange. To prevent this, the re-marketing script provided by the ad exchange should be bound to the designated domain, and at run time the script should verify that it indeed runs within the intended domain.

A few ad exchanges and ad networks provide users the ability to inspect and modify the inferred online interest profile or opt out of personalized ads [14, 19, 20, 21, 22]. However, users have no visibility into how these profiles are generated or used to serve targeted ads [10]. Ad exchanges and ad networks should provide users easy mechanisms to flag suspicious ads they see that are not aligned with their real interests. Additionally, ad exchanges should also encourage users to manually adjust their online interests, and explicitly avoid being targeted in some categories. For example, a user might want to disallow all *Health* related ads. In such a case, a polluter attempting to influence the user's profile with the *Health* category would lead to no ads from this category to be targeted at the user.

A key contributor to the success of the attack is that pollution content immediately impacts the user profile, thus the polluter can almost immediately benefit from the attack. The ad networks can increase the duration between page visits and the impact on the user's profile, thus mitigating the impact of the attack by profiling users interests across a large set of websites visited by the user. However, this delay might be in contrast to

the ad networks' desire for accurate and timely inference of user interests, especially for re-marketing campaigns.

2.5 Related Work

Content Manipulation. The line of work most closely related to our pollution attack is black-hat search engine optimization (bSEO), which manipulate the search engine to promote certain search results. Since bSEO targets the general indexing and ranking process of search engines, the promoted results will be visible to all users. bSEO usually requires sophisticated infrastructure that may consist of hundreds of websites to form a link farm [23]. Building and maintaining these infrastructures require a considerable amount of resources [24]. By contrast, our pollution attack targets the customized content of individual users, which is much easier to launch. On the other hand, techniques that address bSEO are unlikely to be effective against pollution attacks.

Online Advertising Economy and Tracking. The economics of online advertising is discussed in detail in [25], which considers the usage of targeting users based on interests as a key difference between traditional and online advertising. More recently, Gill *et al.* [12] proposed a simple model for capturing the effect of user profile (or "intent") on the revenue obtained by the ad network and the publishers. Using this model the authors stressed the significance of the user profile in the ecosystem by showing that incorporating mechanisms that block tracking, thereby essentially eliminating targeted advertising, can decrease the overall revenue of ad networks by 75%.

In order to build an accurate user profile, ad networks need to track users as they browse the web. Several recent papers measure the extent to which users are being tracked and targeted by ad networks [26, 27, 10]. Rosner *et al.* [27] showed that online tracking of users is ubiquitous and covers a large fraction of a user's browsing behavior. Liu *et al.* [10] focused on Google's DoubleClick network and showed that interest-based targeting is prominent and spans multiple ad categories, with up to 65% of the ad categories received by a user are

targeted based on the user's inferred profile.

In this work we leverage the strong relation between the user interest profile and the economics of online advertising to propose a method for polluting user interest profile for increasing the publisher's revenue.

Fraud in Online Advertising. Fraud in online advertising and countermeasures against these fraud mechanisms have been the focus of a long line of research efforts [28, 29, 30, 31, 32, 11, 33, 34, 35, 36]. The most common fraudulent activities include those where fraudulent publishers leverage click-spam networks or pay-per-view networks to increase the traffic to their sites, and thus increase their ad revenue. Click-spam networks cause fraudulent clicks on ads in order to increase the income of the publisher or sometimes deplete the budget of the advertiser. The most recently study by [32], where the authors conducted a controlled experiment show that click-spam attacks account for 10–25% of the clicks, highlighting the prominence of such attacks. In a recent study, the authors used these results and presented a system [18] that ad networks can use for catching click-spam in search ad networks.

Different from click-spam networks are pay-per-view networks that artificially increase the number of ad impressions of fraudulent publishers by framing the publisher's website within other websites in a camouflaged fashion. Fraudulent activities using pay-per-view networks typically result in impressions that are registered on the camouflaged pages without "genuine user interest" *i.e.*, invalid traffic generation. A recent study [11] have shown a pay-per-view network generates hundreds of millions of fraudulent impressions per day.

Existing online advertising frauds focus solely on increasing the volume of ad clicks or impressions and have largely ignored the impact of user profiles. Our attack complements these existing fraud mechanisms by enabling the publisher to further boost the revenue obtained by participating in either of the networks. Compared to existing fraudulent activities which are suspect to traffic analysis [37, 11], our attack is more resilient against current fraud detection methods.

2.6 Summary

We have presented a new class of attack to pollute a user's profile in a third-party application (service) by embedding content of the third-party application. The attacks exploit the role played by a user's profile in the personalization system of a third-party service. The attacks leverage novel mechanisms to pollute a user's profile to promote certain content on a third-party service. We showed that the profile pollution based attacks are robust against diverse browsing patterns and online interests of users. The attacks are effective in deceiving popular websites to serve content at the attacker's choice and drawing higher-paying ads resulting in a significant increase in ad revenue.

CHAPTER 3

INFERRING USER PROFILE WITH PERSONALIZED MOBILE IN-APP ADS

3.1 Motivation

In-app advertising allows mobile application developers to generate revenue despite publishing their work for free. As in traditional web-based advertising, personalization improves the effectiveness of in-app advertising (and thus increases the revenue earned by app developers). It is well understood that such personalization is only possible if certain user information (*e.g.*, interests, demographic information) is available to the party that serves advertisements, and thus privacy leakage is always a concern. While ad personalization has been well studied for web, relatively little research explores mobile ad personalization in terms of what user information is being collected. We believe research focused on mobile ad personalization is a significant pursuit for the following reasons: 1) Mobile devices are a lot more intimate to users; they are carried around at all times and are being used more and more for sensitive operations like personal communications, dating, banking, *etc.*. Therefore, privacy concerns regarding what information is collected for ad personalization are more serious. 2) Unlike in-browser advertising, where the advertisement content is strictly isolated from the rest of the displayed page by the well-known "Same-origin policy", in-app advertising operates in a new and less understood environment. Thus, in this chapter we try to answer the following two questions which will be of great interest not only to privacy-conscious users of mobile applications, but also to advertisers who try to target specific audience groups:

1. To what degree are in-app advertisements personalized to target different attributes of a user (e.g. interest, demographics)?

2. How much can an app learn about a user by observing personalized advertisements?

To achieve our goals, we collected ground truth demographic data from more than

two hundred real users and tested the correlation between the demographic data with advertisements observed by each volunteer user. This correlation allows us to establish that certain advertisements are statistically more likely to be shown to users of one demographic group than another. We also used the data collected to train models to predict a user's interest/demographic information based on advertisements he/she receives. The accuracy of the generated model indicates how much an ad-hosting app can learn about the user by merely observing the personalized advertisements received.

The work presented in this chapter marks significant improvement in the methodology for studying mobile ad personalization as well as an extension in scope for such studies. Specifically, our work is the first that is based on ground truth data collected from real users, while prior work [38, 39] mostly study advertisements received by synthetic users that are expected to have certain interests or belong to certain demographic groups. We argue that results obtained using synthesized users can never be conclusive if we do not know how the studied ad-network builds user profiles. For example, while one may try to make a user appear like a middle aged white male to the advertiser/ad network by downloading and running apps that are predominately used by the target population, one cannot know for sure that this is actually a signal that the advertiser/ad network is listening for or whether this signal is strong enough for the advertiser to conclude the user is a middle-aged white male.

Previous studies have shown the possibility of privacy leakage through web advertising [40, 41]. We study whether the environment of in-app personalized mobile ads presents new privacy threats. Specifically, we investigate whether users' demographic profile can be reconstructed based on the ad content delivered to mobile apps. Ideally, as the case of in-browser advertising, personalized ads are delivered directly to users in the iframe of ad networks, and thus only users can know the ad content personalized on their information. However, unlike web advertising, mobile in-app advertising allows app developers to access users' personalized ads. These ads might reflect users' real interests and other demographics, because the ads run in the same process space of the app.

To maintain a reasonable scope for our work, we focus on the Android platform and Google's mobile ad network - AdMob. Given the major market share of Android (76.6%) in current mobile shipments and that of AdMob (35%) on Android devices [42, 43], we believe they are very good representations for studying mobile advertisements. We note that even though we only studied one ad network, the same methodology can be applied to study other ad networks and to determine whether/how much other user data (e.g. sexual orientation) is effectively used in ad personalization and might be leaked to apps hosting those ads. The amount of leaked personal data depends on the degree of personalization of the studied ad-network, *e.g.*, adversaries are expected to learn less or even no personal information from a less sophisticated ad network than from Google's ad network.

3.2 Background

In this section, we briefly describe the ecosystem of mobile advertising and its related targeting mechanisms. We also discuss the differences between web advertising and mobile advertising that cause potential privacy leakage on mobile platforms.

3.2.1 Ecosystem of Mobile Advertising

Publishers, advertisers, and ad networks are the three main components of both web and mobile advertising. The only difference between web and mobile advertising is that in web advertising, the only kind of publishers are owners of websites, while in the mobile case, publishers can also be developers of apps who might spare some of the screen real-estate for in-app ads (e.g. banner). Advertisers, on the other hand, set up ad campaigns to show their ads to specific users in apps if requested by the publishers. In return, the advertisers pay the publishers for serving their ads, which might potentially generate more transactions later from users if they are interested. In order for the publishers to connect with the advertisers, ad networks are formed. By partnering with millions of publishers, it is possible for the ad networks to integrate user information contributed by participating apps, generate profiles to

predict various attributes of the user (e.g. age, gender, income level) and use these profiles to push targeted ad campaigns from advertisers to certain group of users. Such data collection and profile building is of paramount importance to all three parties, since accurate targeting is crucial for both effectiveness of ad delivery [2] and increasing publisher revenue [3].

3.2.2 Targeting in Mobile Advertising

Ad networks can monitor app activities, app lists, device models, etc. on mobile devices to automatically collect and infer the users' demographic and interest profiles. Information like demographics, geo-locations, etc. can also be provided from app developers through ad control APIs [44] for better quality targeting in order to maintain a higher click-through rate of ads, resulting in higher revenue. On major platforms like Android, since most users would login to their Google account before starting to use the devices, more personal information can be gathered from these accounts. With all the potential paths for information collection, an ad network is able to use these personal features to create/update user profiles, and push personalized in-app ads to targeted users.

We have studied the interface provided by major ad networks (*e.g.*, Google) for advertisers to specify their target population, and concluded ad networks generally provide the following three types of targeting: topic targeting, interest targeting and demographic targeting. Such offerings to the advertisers suggest that the ad networks have at least some estimate for each user regarding the attributes that can be used for targeting.

Topic Targeting. Topic targeting lets advertisers place their mobile ads in apps that are related to the ad content. Simply by selecting one or more ad topics through an ad network interface, advertisers can have the ad network deliver to apps that are relevant. For example, by targeting the "Autos & Vehicles" topic, advertisers can ensure that auto-related ads are pushed to apps that include content about cars or other automotive themes. More precise subtopics, such as "Truck & SUVs", are also included in the general topic of "Autos & Vehicles" to achieve more effective topic targeting [45].

Interest Targeting. Interest targeting involves reaching to users interested in products and services similar to those advertisers offer, even when they are using apps that are not directly related to the products or services that are advertised. The interest profiles of users can be pre-built by the ad network, based on users' usage patterns on mobile devices, ad categories that they have clicked on before, and so on. Cross-platform correlation for interest profile might also be necessary for locating the same user across PC and mobiles. By having advertisers choose the interest categories, the ad network can advertise to those who have shown interests in the same categories before in their profiles [46].

Demographic Targeting. Advertisers use demographic targeting to deliver ads to users who are within a chosen demographic group. For example, if the advertised business caters to a specific set of users within a particular age range (e.g. younger people like sport cars better), then targeted ads to that group of people are more effective than others [47].

3.2.3 (Lack of) Isolation for In-App Advertising

In web advertising, the in-browser ads are usually delivered directly in iframe from ad networks to users [10]. These ads on websites are isolated from publishers of websites in terms of its content and code due to the Same Origin Policy (SOP). Thus, usually only the users will be able to view the ad content that is personalized by ad networks based on their collected personal information. In-app advertising however, has targeted ads running in the same processes as the apps themselves with the same permission level. Therefore, all app developers are able to access to users' personalized ads in their own apps, which can be reverse-engineered to show users' real interests and demographics. In fact, Shekhar *et al.* have also mentioned the necessity of separating processes between an application and its advertising for security purposes [48]. In the current study, we are examining the same argument from the privacy perspective.

Even though a recent report shows that Google has considered utilizing HTTPS protocol to encrypt ad-related traffic [49], we argue that the protection is not useful regarding current

privacy concerns. Since encryption can only protect at the level of communication channel between apps and ad networks, ad content is in plain text at the time received ads are being displayed to users. Hence, app developers can still access the targeted ads delivered to their apps in the decrypted form.

3.3 Methodology

In this section, we first describe our research problem, then discuss the challenges and outline our approaches.

3.3.1 Goals of Study

We seek to answer two key questions regarding user privacy in personalized mobile advertising:

1. What personal information about real end users can a dominant mobile ad provider such as Google know and use in personalized mobile advertising? Specifically, we want to understand how much mobile ad providers know about real users and how that knowledge regarding real users is exploited for providing personalized ads.

2. Could personalized mobile in-app ads be served as a channel of private user information leakage? More specifically, could an adversary (i.e. mobile app developer) with access to personalized mobile ads gain any information about real users?

To answer the above questions, we need to clearly define private user information on mobile devices with respect to personalized advertising. We study two classes of personal information in our work:

Interest Profile. A user interest profile models a user's behavior on the web or/and on the phone and is built by online trackers and ad providers. It consists of labels of tens or even hundreds of interest categories. Interest targeting in mobile advertising targets users who match the combination of interest categories specified by the advertiser. For instance, Bob

is a fan of video games and he spends 2 hours playing games on his smart phone each day. Bob also reads many articles about sports news through specific applications on his phone. A interest profile like $\{Games, Sports\}$ well represents Bob's interest. A developer of a new basketball game may ask ad providers like Google to target users that have interest profiles similar to Bob's. Bob may see and click an ad of the basketball game and then become a user of this game.

Demographics. In recent years, ad providers have started to provide a more sophisticated targeting option - demographic targeting - for advertisers. For example, advertisers can target users by gender, age and parental status on Google AdWords [50]. This indicates that ad providers are actively tracking and modeling private personal information other than interests. Google has confidently shown its knowledge of user's gender, age and parental status in its personalized service. This raises the question of what other personal information online trackers are trying to learn from their users, which concerns both consumers and policy makers. In this study, we examine the following demographic categories: *Age, Gender, Education, Income, Ethnicity, Political Affiliation, Religion, Marital Status, and Parental Status*.

3.3.2 Challenges and Our Proposed Approaches

In the process of designing our experiment to determine which information is used by Google for personalizing advertisements, we have identified two challenges that any similar experiment will need to overcome.

Triggering personalization based on target attributes. To determine whether certain user information is collected and used for advertisement personalization, we need to devise a method to "provide" the ad network with that piece of user information. For example, if we want to determine whether the user's gender is used for advertisement personalization, we need to make sure that *if gender is indeed used, the ad network should have high confidence in its estimation of the user's gender when it is serving ads to a user under observation*.

A previous approach to answering this question was to build artificial user profiles by performing certain actions that are believed to be observed by the ad network (e.g. installing certain apps that are predominately installed by one gender but not the other).

We find this approach circular in nature. In particular, if we are trying to determine *what* user information is used in personalization, we must assume we do not know *how* the ad network deduces the personal information by observing the user's behavior. In fact, even the set of user behaviors observed/used by the ad network to form the user's profile is generally unknown to us. As such, there is no reliable way for us to say, for example, "if the ad network is providing gender based personalization, it must have concluded that the user under observation is a male after we have performed these operations". In other words, if our experiments based on synthesized user behavior come back negative, we cannot tell if that is because the studied user attribute is not used for personalization or if it is because of flaws in the profile synthesis process.

In this work, we overcame the above problem by recruiting real users and collecting their demographic information as well as the personalized advertisements observed by these users. By having ground truth from real users, negative results that show no difference in advertisements observed by people of different demographics can be concluded as "the ad network failed to provide advertisement personalization based on that piece of demographic information".

Isolate personalization from non-target attributes. A related problem we face in trying to determine if certain user information is used in as personalization is controlling for the other factors that are known to be used in ad personalization. For example, in in-browser advertising, we know that the user's geo-location is an important factor in determining which ads he/she sees. Similarly, if we collect advertisements seen by different users on apps they installed, the difference in the ads they receive may not be based on demographic differences; rather, they may be caused by the categories of ads requested by different apps using ad control API. From our experience in designing similar experiments, we are certain that such

noise must be eliminated if we are to draw any statistically significant conclusion confirming the existence of personalization based on any user attributes that are not previously known to be used for personalization. To this end, we chose to collect personalized advertisements seen by different users with our own tailored app that does not employ any ad control API, and always send requests for advertisements from our own IP address.

3.3.3 Experiment Design

In this subsection, we present the details of our experiments, which was approved by our Institute Review Board (IRB).

Subject recruitment. We recruited Android users located in the United States from Amazon Mechanical Turk [51] as subjects to complete surveys regarding the subject's demographics and interests. Each subject was also required to install our Android app for ad traffic collection. Using the surveys, we are able to gather ground truth about end users for evaluating mobile ad personalization. This eliminates the artifact effect of building synthesized user profiles. To ensure users pay attention to the survey, we inserted some trick multiple choice questions in random order. They are considered simple to solve and require no more than basic skills (e.g. 1+1=?). Subjects' answers to the survey would be rejected if they failed to complete those trick questions correctly. The survey questions and multiple choice questions were also randomized in order, for the purpose of removing potential biases from subjects' responses. Using survey responses along with the ads collected in our app, we were able to analyze the relationship between personal user information and mobile ads.

Ad collection. To isolate the impact of application-based targeting, we designed a blank Android app dedicated for collecting mobile ads. The app initiates 100 ad requests to ad networks without setting any targeting attributes. We selected Google AdMob as our target for ad data collection, due to Google's dominance in the mobile advertising industry. Since location targeting is prevalent in practice, we established a secure VPN connection to our server from the user's device to isolate impact of the location. In particular, all Internet

traffic through all the apps installed on subject's phone was tunneled through the VPN service provided by our App, but we only collected advertising traffic generated by our App during the data collection phase. To avoid collecting ads intended for applications other than ours, we instructed our Mechanical Turk subjects to keep our app running without operating other apps until the app finished data collection and turns off the VPN tunnel. The entire data collection process took about 2-15 minutes, depending on the network condition of the user's device.

We note that the VPN tunneling employed in the ad collection process makes all ad requests from our blank app originate from our IP addresses. As a result, ads collected by our app are personalized for users at our location, instead of the individual subject's real location. While this can be considered a kind of noise we inject into our data, we argue that this is also an advantage of our experimental design: we can eliminate the influence of geo-location on ad profiling, and better study how the other aspects of a user's profile (demographics, interest) affect the ads she/he receives. Furthermore, if geo-location were included in the study, we would need a significantly higher number of volunteers to cancel out its influence.

We believe that our experiment had limited impact on the subject's ad profile for two reasons: 1) we believe a subject's profile is built upon long term observation of how he/she interacts with his/her phone/apps, and thus any influence we introduce over a short interval with 100 ad requests will be insignificant, and 2) "blank" ad requests (i.e. requests that do not specify any information regarding the intended audience) from a blank app, with no clicks on the received ads, should present the ad network with no useful signals for updating the subject's profile.

Landing URL extraction. We use the landing URL (the destination URL that a user agent will be redirected to after clicking/touching an ad) as the representation of an ad in our analysis. Specifically, we tried to extract the landing URL directly from the meta data contained in the HTML source of an ad. Through our analysis on the HTML sources

representing ads, we identified several attributes and keywords that helped us extract the landing URL of an ad. The attributes and keywords we used for landing URL extraction were *buildRhTextAd, adurl=, final_destination_url, destination_url, destinationUrl, click_url*, and *go.href*. For ads that we were not able to extract a valid landing URL, we replayed the ads on our server by reopening the HTML source in a desktop web browser and clicking through the ad to reach the final landing page. We also replayed some ads that the extracted landing URL directs to some known ad networks, which usually further relay the user's visit to the final destination or other ad networks. In the end, we had to replay only 2,372 out of 39,671 (5.98%) ads in our dataset to get the final landing URL; thus, our approach allowed us to minimize the impact of our study on the mobile ad ecosystem.

Landing URL post processing. The landing URLs extracted using the above approach usually contain a long list of tail attributes, which are used to identify the source of the visit, *e.g.* creative_id, campaign_id , mobile app name, and ad network. We cluster the landing URLs by removing those tail attributes and grouping them into ad campaigns. Ad campaigns are further merged if they share the same domain and prefix but different resource names, except for Google Play Store apps which all start with https://play.google.com/store/apps/details?.

Ad categorization. Each ad (landing URL) is categorized into one of the 24 root interest categories that Google provides for as targeting options. If an ad is of a Play Store app (identified by the above URL pattern), we use a script to directly extract its corresponding interest category from its Play Store web page. For the other ads, we rely on Google Ad Preference [52] for labelling. In particular, we built a Google Ad Preference crawler using a headless browser - PhantomJS [53]. We crawled the interests that were generated by Google right after visiting each ad landing URL for a consecutive of 10 times starting with a blank browser profile. Such interests were used to label the ads correspondingly. Yahoo Content Analysis API [54] is also used in cases where we do not get any assigned categories on Google Ad Preference page from the previous method. All the Yahoo categories are mapped

to their closest representation in the 24 Google root interest categories. For those ads that cannot be automatically categorized by the above steps, we manually assign the category that best matches the content of the ad landing page. We had three persons categorizing the mobile ads independently. If the decisions on an ad from the three human labellers conflict, a final label is selected based on mutual agreement.

Ad-containing packages detection. Ad networks employ information like the list of installed apps and apps a user has used to infer his/her personal information [47]. Specifically, all interaction with one ad network will be captured through apps that contain the code of the ad network. Ad networks' libraries are typically called in the same manner by different developers. The UI elements that are used for renderring ads have the same name or identifier across different applications. This enables one to learn the applications that include the same ad library on a user's mobile device. We detected all packages that include Google's AdMob library through the Android PackageManager the user has installed. Such information is helpful in understanding the profiling mechanism of black-box ad networks. We used the list of ad-containing packages as one class of features in our privacy evaluation in Section 3.5.

3.4 Characterization of Mobile Ad Personalization

In this section, we first present details of our dataset collected from 217 real users, then study the correlation between user's demographic and interest profiles and the personalized mobile ads.

3.4.1 Dataset

We created a Human Intelligence Task (HIT) on Amazon Mechanical Turk for workers located in the United States. Each worker was asked to install our app, which will collect 100 ads when executed. After the completion of the data collection process, the worker was asked to fill out a survey regarding his or her interests and demographic information (as defined in Section 3.3.1). We had run our HIT for 12 days on Amazon Mechanical Turk. In

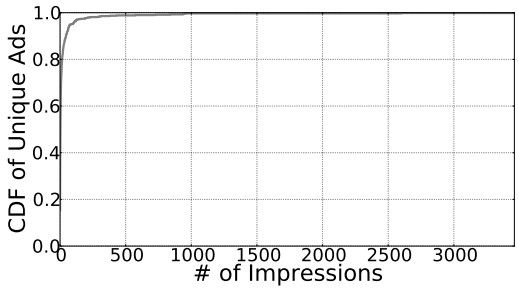


Figure 3.1: Impression distribution of unique ads.

the first 3 days of our experiment, each worker was compensated for $1.00. To attract more workers to work on our HIT, we increased the reward amount to $1.25 on the 4th day.

In total, we were able to collect survey responses from 284 users and successfully collected 100 ads for each of 217 users. The other 67 users quit before the ad collection was complete, and we discarded the partial data collected. Table 3.1 shows the distribution of the demographics of the 217 users in our dataset. Out of 39,671 ad impressions we captured, 33,135 ad impressions for 695 unique ads were issued from the 217 users. We observed that some users had run our data collection application multiple times, and for these users, we only used the first 100 ads collected. Figure 3.1 displays the distribution of the 695 unique ads in terms of ad impressions. Surprisingly, two ads (*Zoosk - #1 Dating App*[1]: 3,461, *Samsung for Business*[2]: 2,602) accounted for 28% of total ad impressions in our dataset. In the 695 unique ads, 500 (72%) are of applications on the Google Play Store, which generate 9,124 impressions (42%). The remaining 195 (28%) ads contribute to 12,576 impressions (58%). Figure 3.2 shows the distribution of the 695 unique ads in terms of number of users. Five ads were delivered to more than 50% of the 217 users. Figure 3.3 gives the number of unique ads displayed to each user, Figure 3.4 breaks down the ad impressions into interest categories, and Figure 3.5 presents the number of users that have received ads in each category.

[1]https://play.google.com/store/apps/details?id=com.zoosk.zoosk
[2]http://www.samsung.com/us/business/samsung-for-enterprise/

Table 3.1: Demographics distribution of subjects.

Gender		Political Affiliation			Parental Status	
Female	Male	Independent	Democrat	Republican	Not a parent	Parent
95	122	108	80	29	128	89
43.78%	56.22%	49.77%	36.87%	13.36%	58.99%	41.01%

Income			Marital Status		
$< \$30K$	\$30K-\$60K	$> \$60K$	Single	Married	Separated, divorced or widowed
107	67	43	124	73	20
49.31%	30.87%	19.82%	57.14%	33.64%	9.22%

Religion			Age				
Atheist	Non-Christian	Christian	18-24	25-34	35-44	45-54	55+
82	47	88	45	106	47	14	5
37.79%	21.66%	40.55%	20.74%	48.85%	21.66%	6.45%	2.30%

Ethnicity				
Other	Hispanic	Asian	African American	Caucasian
8	12	12	23	162
3.69%	5.53%	5.53%	10.60%	74.65%

Education			
High school or less	Associates	Bachelor	Master or higher
78	50	71	18
35.94%	23.04%	32.72%	8.30%

3.4.2 Interest Profile Based Personalization

In this subsection, we study interest profile based personalization. Specifically, we try to understand how well ad networks learn about the real users' interests.

We use $P_{user,i}$ as the real user interest profile derived from the survey response from user i. The ad interest profile $P_{ad,i}$ is defined as the set of interest categories of all ads delivered to user i. We use the following three metrics to evaluate the similarity between real user interest profile and ad interest profile.

1. Size of an interest profile, which is number of categories in each interest profile.

2. Precision, which is defined as $|P_{user,i} \cap P_{ad,i}|/|P_{ad,i}|$. Precision represents the fraction of categories in an ad interest profile that match the user's real interest profile. It measures how precisely ad networks know user's real interests.

3. Recall, which is defined as $|P_{user,i} \cap P_{ad,i}|/|P_{user,i}|$. Recall is the fraction of categories

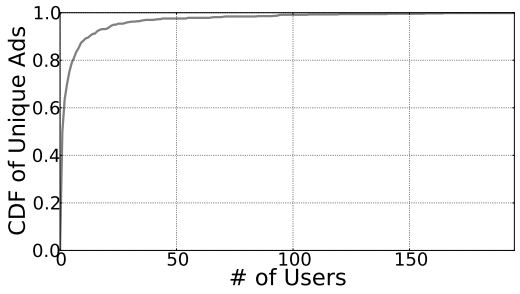


Figure 3.2: User distribution of unique ads.

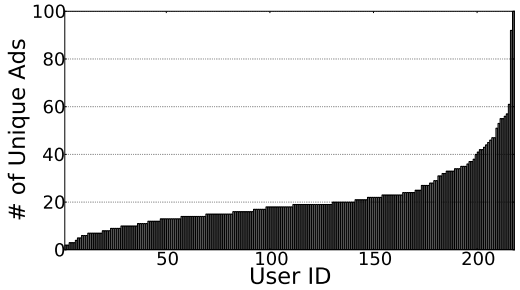


Figure 3.3: Number of unique ads of each user.

in the real user interest profile that are presented in the ad interest profile. It represents the ad network's coverage of the users' real interests.

We first show the sizes of real user interest profile and ad interest profile for each user in Figure 3.6. The sizes of the two interest profiles vary significantly across users, indicating a diverse distribution of user interests. As we can see in Figure 3.6, there is no clear correlation between the sizes of the two profiles, suggesting size of interest profile is not a good metric for evaluating the similarity between the two profiles.

The distributions of Precision and Recall are shown in Figure 3.7 and Figure 3.8, respectively. For over 79% of the users, at least 21% of the categories in the ad interest profiles are correct. For 11% of the users, at least 83% of the categories in the ad interest profiles are correct. In terms of Recall, Google could cover at least half of real user interests for 60% of the users. The results demonstrate that ad networks like Google can build accurate interest profiles of mobile users and use the profiles built for personalizing mobile in-app advertisements.

To understand how many ads are personalized based on real user interests, we further

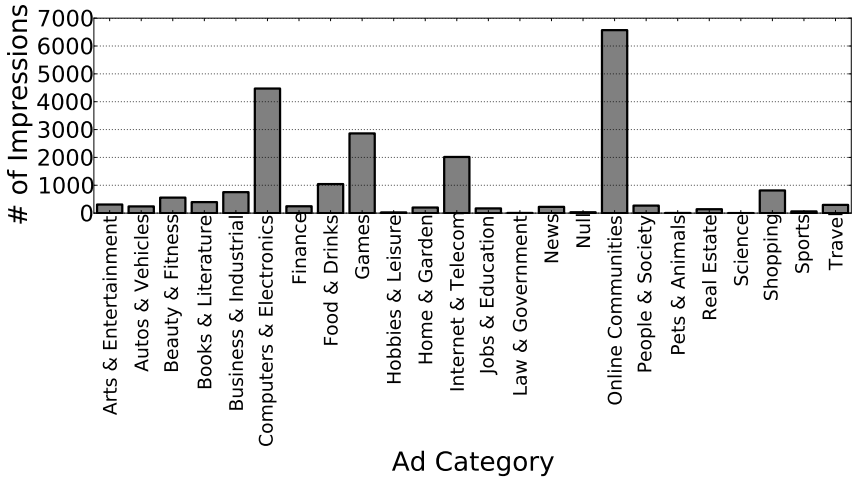


Figure 3.4: Number of ad impressions in interest categories.

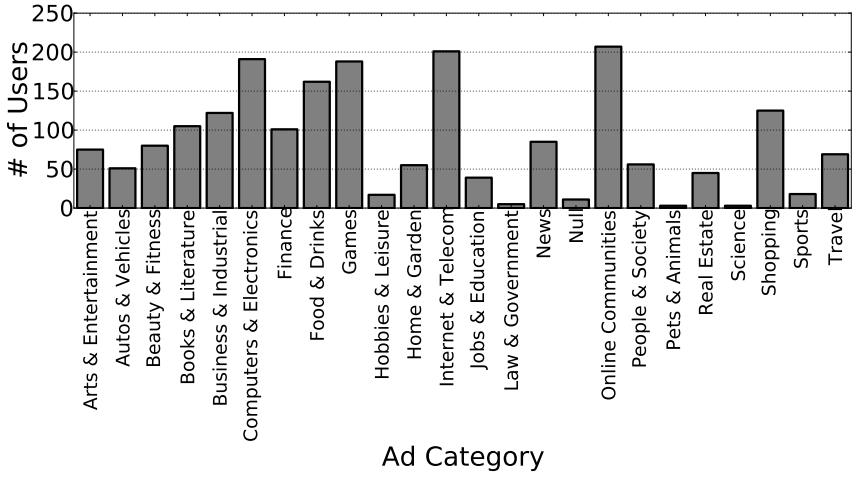


Figure 3.5: Number of users in interest categories.

looked for ads that match real user interests, which we refer to as *precise ads*. Figure 3.9 displays the number of precise ad impressions of the users. The result suggests that Google is actively personalizing a large fraction of its ad deliveries. For 41% of users, more than 57% of their ad impressions match their real interests.

Summary. Our analysis shows that mobile ads are highly personalized based on user interests. The ad interest profiles derived from observation of 100 ad impressions are quite close to users' real interest profiles with good precision and recall. More than 83% of the categories in the ad interest profiles are correct for 11% of users, and more than 50% of real user interest categories are covered in ad interest profiles for 60% of users. We further find a

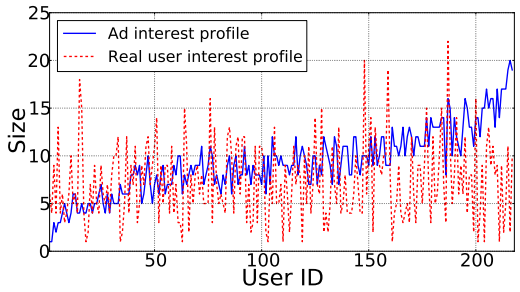


Figure 3.6: Number of interest categories in real user interest profile and ad interest profile.

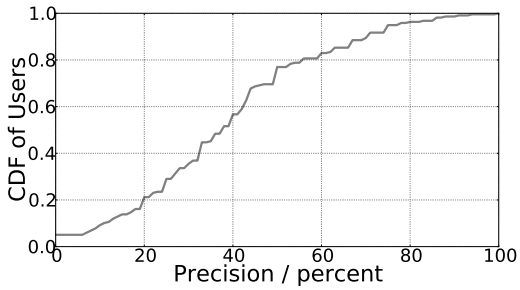


Figure 3.7: Precision distribution of user profiles.

large fraction of mobile ads match with real user interests. More than 57% of ad impressions for 41% of users fit users' real interests.

3.4.3 Demographics Based Personalization

As discussed in Section 3.2, since ad networks allow advertisers to target their ads towards specific demographic groups, we strongly believe the ad networks already have profiles that capture various demographic information of the users. In this subsection, we seek to quantify to what extent real users' demographics may have been used for ad personalization. Note that as shown in [50, 47], gender, age and parental status are the only 3 demographic categories that Google explicitly allows advertisers to use for targeting purpose. Thus our observation of strong correlations between ads and other categories of demographic information may not be the results of explicit ad targeting. However, for the sake of brevity, in the following discussion we will attribute strong correlation between ads and demographic information to (possibly unintended) personalization.

We grouped the 217 users into different demographic sets for each demographic cate-

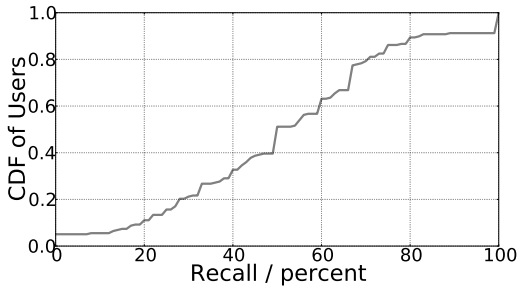


Figure 3.8: Recall distribution of user profiles.

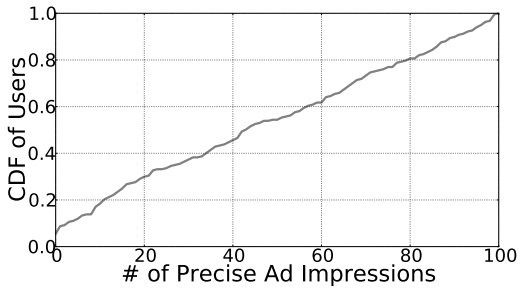


Figure 3.9: Number of precise ad impressions of users.

gory. We employed statistical tests to determine whether one ad is correlated with a given demographic category. Specifically, we counted the number of times an ad is shown to users in different demographic groups. Then the ad was tested for independence with the demographic category by using Pearson's chi-squared test. We excluded ads with an expected number of impressions fewer than 5 for any demographic group, which is a common practice when applying Pearson' chi-squared test. The null hypothesis is that the ad being tested is independent of the demographic category. We set the significance level for our tests to be 0.005. If the p-value of one ad is less than the significance level, we reject the null hypothesis and label the ad as personalized based on (correlated with) the demographic category under test.

The number of unique personalized ads in each demographic category is presented in Figure 3.10. It is not surprising that many ads are targeting users by gender. For example, the ad for the game *Game of War - Fire Age*[3] is shown to 66 males (70%) for 614 impressions (64%), while only 28 females (30%) received the remaining 342 impressions (36%). On

[3]https://play.google.com/store/apps/details?id=com.machinezone.gow

the other hand, the ad impressions of the game *Cookie Jam*[4] are dominated by female users. 182 impressions (96%) were shown to 13 females, while 4 males share the remaining 7 impressions (4%). There are even 33 ads that are exclusive for one gender class.

Parental status is the second most personalized demographic category in our result. Ads of social applications and websites like *Zoosk - #1 Dating App* and *Facebook*[5] are leaning toward users that are not a parent. 2,312 ad impressions (67%) of Zoosk and 403 ad impressions (70%) of Facebook are shown to non-parent users. Interestingly for Marital Status, we found social applications and websites also show preference to non-married groups.

We are surprised that there are many ads that are dependent on users' income level. In our dataset we found ads of many games (*e.g.*, *FINAL FANTASY Record Keeper*[6] : 71%, *Cookie Jam* : 69%, *League of Angels -Fire Raiders*[7] : 67%, *World Series of Poker - WSOP*[8] : 67%) are shown more toward users in our low income group (with annual gross income below $30,000). We could not find any income targeting option on Google AdWords and Google AdMob for advertisers. We do not think advertisers on Google currently are explicitly targeting users by income. The result suggests that Google may tailor ad deliveries based on users' income as a result of its personalization algorithms to further increase click-through-rate or conversion rate, which we cannot prove for sure. However, the practice of ad syndication makes it possible that those impressions may be purchased through other ad networks that offer income targeting and other demographics targeting [55]. It is also possible that the correlation with income we observed is a result of income's correlation with age, parental status, or other demographic information. For example, in our dataset older people who have children generally have higher incomes.

Our statistical test suggests 58 unique ads are correlated with users' religion. However,

[4]https://play.google.com/store/apps/details?id=air.com.sgn.cookiejam.gp
[5]https://www.facebook.com/r.php
[6]https://play.google.com/store/apps/details?id=com.dena.west.FFRK
[7]https://play.google.com/store/apps/details?id=com.gtarcade.loa.ph
[8]https://play.google.com/store/apps/details?id=com.playtika.wsop.gp

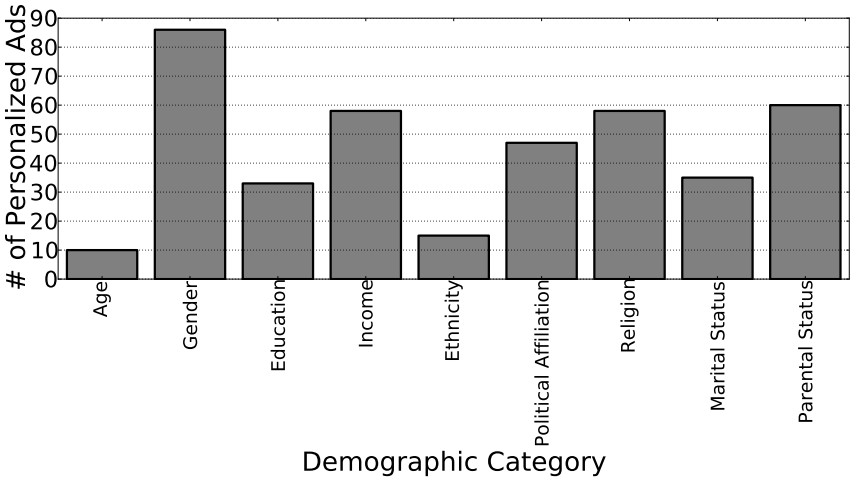


Figure 3.10: Number of unique ads that are personalized based on demographics.

we found only one ad, *Peace With God*[9], which is clearly related to religion. 57 impressions (57%) of this ad are shown to Christians. We conjecture carefully that the dependencies of the other 57 ads may be a result of religion's correlation with other demographic categories. Similarly, we could not find any explicit evidence of targeting by Political Affiliation. We cannot explain the correlations of ads with users' Political Affiliation due to a lack of insight into ad networks' secret personalization algorithms.

For age, education and ethnicity, we observed personalization in lower degrees in terms of number of correlated ads. However, Figure 3.11 gives a different view of demographics based personalization. We present the number of impressions of the personalized ads in each demographic category in Figure 3.11. Except for gender, all demographic categories have a number of personalized ad impressions greater than 10,000 (46%). Although there are fewer personalized ads in some demographic categories, the effects of personalized ads in these categories are not significantly lower than those of personalized ads in other categories.

We further quantify the impact of demographics based personalization on individual users. Figure 3.12 and Figure 3.13 show the distribution of users in terms of number of demographically personalized ads and number of demographically personalized ad impressions they receive, respectively. 76% of the users have received at least 10 ads that

[9]http://peacewithgod.net/mobile/

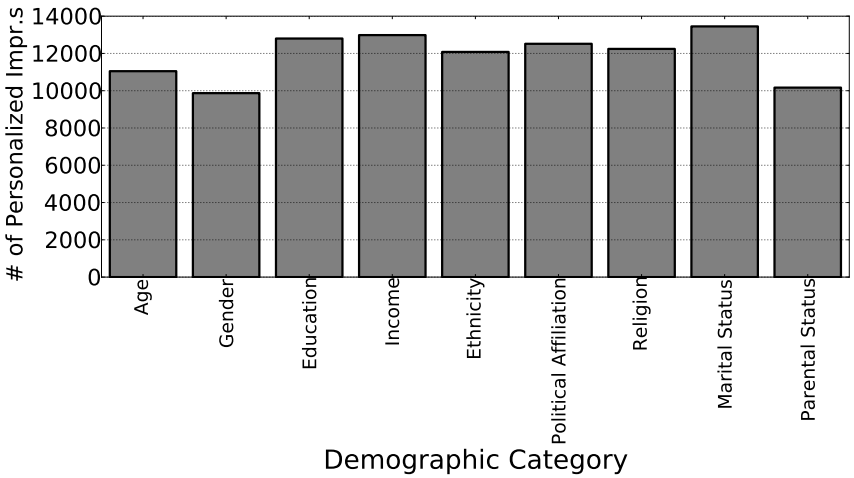


Figure 3.11: Number of ad impressions that are personalized based on demographics.

are personalized based on demographics. Surprisingly, at least 73 out of the 100 impressions are of ads that are personalized based on demographics for 92% of the users. Note that the values derived here include personalized ads and impressions displayed to demographic groups that are not the primary targets of the ads.

Summary. Using statistical tests, we found ads that are personalized based on (correlated with) demographics. Gender is the demographic category that we observed the highest number of unique personalized ads. Personalization may be an explicit targeting option expressed by advertisers (age, gender and parental status), or it may be result of an ad network's proprietary personalization algorithms (income, religion, etc.). Our results in Section 3.5 will shed more light on our observation about demographics based personalization.

We also found that demographics based personalization in mobile advertising is prevalent in practice. 76% of our users have received at least 10 demographically personalized ads, and more than 73% of ad impressions of 92% of users are demographically personalized. Ads that are delivered exclusively to certain demographic groups are highly indicative of a real user's personal information. Together with non-exclusive ads that are correlated with some demographic categories, those ads may be a good representation of real user's demographic profile. This raises a great opportunity for adversaries to learn the private personal information of real users. As we will demonstrate in Section 3.5, personalized ads

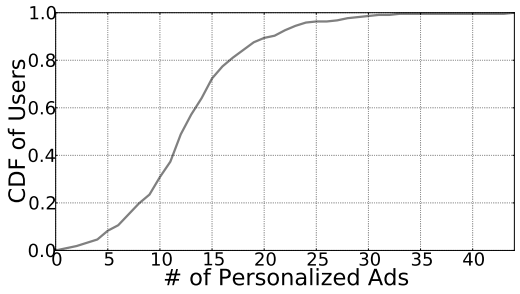


Figure 3.12: Number of unique ads that are personalized based on demographics across users.

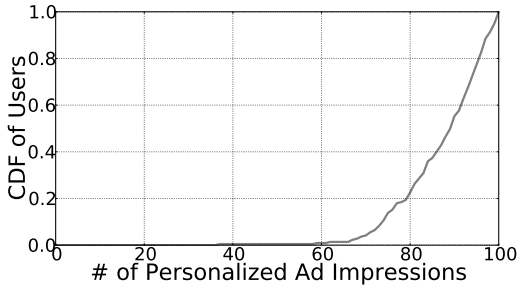


Figure 3.13: Number of ad impressions that are personalized based on demographics across users.

serve as a new channel for privacy leakage on mobile devices.

3.4.4 Comparison with Previous Studies

Previous studies showed that demographic information of users was not commonly used for mobile in-app ad personalization (only found on Google's ad network), and the user's geo-location played a significantly more important role than his/her interests/demographics in determining what in-app advertisements he/she is receiving [38]. In contrast, our results illustrate that both demographic and interest profiles of users have a statistically significant impact on how in-app ads are selected for the same ad network studied (We did not study geo-location as discussed in Section 3.3). Some explanations for the discrepancy between the results across studies are as followed.

First of all, what [38] measured to show the significance of demographics information in ad personalization is very different from what we are measuring. In particular, most of the results in [38] used keywords/ad-control attributes in ad requests generated by applications to

measure *how often demographics based ad personalization is requested by apps*. However, in this chapter, we studied *how often one's demographics information strongly correlates with the ads he/she receives* and use it as an indication of successful demographics-based ad personalization. In other words, [38] measured personalization requested by the app developers, while we looked for evidence of successful personalization performed by the ad network, which may have a significantly better ability to profile a user than app developers.

Secondly, instead of using synthesized user profiles to harvest mobile in-app ads as in the previous studies, we collected ads using real user profiles, which led us to observe more ads that are correlated with user's interest and demographics. We believe that authentic user profiles could generate the right signals to trigger ad personalization, while synthesized user profiles might not. In addition, we studied more complete demographic profiles from users than previous ones, which enabled us to discover new types of demographics based targeting that were not observable in previous studies. We believe that our findings compliment those from previous studies and helps the research community better understand mobile in-app ad personalization.

3.5 Privacy Leakage through Personalized Mobile Ads

As shown in Section 3.4, mobile ads are highly personalized based on user interests and demographics. As a result, a set of ads collected from a user's device can be seen as an accurate representation of that user's real personal information, potentially including sensitive data. We have already shown in our real user study that the ad interest profiles inferred from 100 ad impressions closely match with real user interest profiles. It is also possible to infer user's demographics from personalized mobile ads, as we will demonstrate in the following subsections. The rest of this section is organized as followed: we first discuss the feasibility of privacy leakage through personalized mobile ads in Section 3.5.1; our experiment setup and definition of evaluation metrics will be presented in Section 3.5.2; finally, in Section 3.5.3, we present the evaluation results. Section 3.5.4 discusses the privacy

implication from our experiment.

3.5.1 Technical Feasibility

Previous studies have shown the possibility of reconstructing user interest profiles in web advertising under the threat model that the adversary can eavesdrop on a victim's unencrypted ad traffic [40]. To provide stronger security in serving online advertisements, the online advertising industry is taking steps towards enhancing the security of ad transmission through HTTPS protocol [49, 56]. The adoption of HTTPS in ad serving could certainly defeat the above attack in web advertising. However, we argue that in the scenario of personalized mobile advertising, the threat to user privacy is much greater even with the protection that HTTPS provides.

In contrast with web advertising where the personalized ad contents are protected from publishers and other third-parties by the Same Origin Policy, there is no isolation of personalized ad contents from the application developers on a mobile platform such as Android. As such, an adversary does not need to sniff the ad traffic of a victim mobile device user. Even when HTTPS is enforced, any host application can still read the ad contents displayed within it regardless of whether encryption is enabled during data transmission. The ability of observing personalized ads on mobile devices opens a new attack vector for private personal information leakage, which we will demonstrate next.

3.5.2 Demographics Learning from Personalized Mobile Ads

In this section, we will try to determine if the same can be done for the user's demographics information, which is potentially more sensitive than the user's interests. In particular, we applied machine learning algorithms to build models for predicting a user's demographic information (for all studied categories) based on the ads he/she has seen, and evaluated the accuracy of the generated models.

We use the combination of the number of impressions of ads that are correlated with each

demographic category and the list of installed apps that contain the Google AdMob library as features. Each sample is labeled with one class in the corresponding demographic category according to the survey response. We tested a set of basic classification algorithms (Decision Tree, Logistic Regression, Multinomial Naive Bayes, K-Nearest Neighbors, Random Forest, SVM) to estimate our ability to learn users' demographic info. A dummy classifier that predicts by randomly guessing was used as the baseline classifier for comparison.

The classifiers are implemented using the *scikit-learn* package [57] of Python. All classifiers are evaluated with 5-fold cross validation to avoid overfitting. Specifically, the 217 samples are randomly divided into 5 different sets (folds), and for each fold, the other 4 folds of samples are used as training set to train the model. The resulting model is then validated using the remaining fold as test set. For the sake of fairness, all classification algorithms are cross-validated using the same division of 5 folds.

We used the accuracy of the prediction as the metric for measuring the severity of privacy leakage through personalized mobile ads. We define the accuracy of a classification model as the number of accurate predictions divided by number of all predictions. Note that one prediction is accurate only when the predicted class is exactly the same as the label. Thus for binary classification problems (*e.g.*, gender and parental status) the dummy classifier has accuracy of 50%. For multi-class classification problems, which are harder than binary classification problems, the accuracy of dummy classifier is 1 divided by number of possible classes. We report the cross validated accuracy (the mean accuracy of the 5 validations) as the accuracy of one classifier.

A point worth emphasizing is that in a perfectly safe/privacy-preserving system, the adversary should have no advantage in knowing victims' personal information, *i.e.* the adversary cannot have better accuracy than that obtained from tossing coins. Thus if the accuracy of an adversary's model is significantly higher than that of the dummy classifier, it suggests that the adversary has significant advantage in learning victims' personal information. Our goal is to understand the possibility of privacy leakage in personalized mobile ads,

thus any result that is above the baseline accuracy (the accuracy of the dummy classifier) should be considered as a potential source of privacy leakage. We present our findings in next part of this section.

3.5.3 Evaluation

Table 3.2 lists the accuracy of all the classifiers and the accuracy of the dummy classifier for each demographic category. The cell in bold represents the highest accuracy score in each column. Overall, SVM performs the best in predicting all demographic categories. For all demographic categories, we could find at least one classifier that performs much better than the dummy classifier. This is particularly true for gender, that three classifiers (Logistic Regression, Multinomial Naive Bayes, and SVM) are able to predict accurately for over 70% of the users. The same three classifiers also perform well for parental status, with accuracies above 65%. Surprisingly, four classifiers are able to accurately predict the ethnicity of more than 70% of users, in contrast with the 20% accuracy of the dummy classifier, but this can be attributed to the distribution of our sample. By comparing with the result in Table 3.1, we find that the distribution of our subjects is highly biased in terms of ethnicity. 74.7% of our subjects are Caucasians, and according to the United States Census Bureau, 72.4% of the U.S. population was Caucasian in 2010 [58]. With this knowledge, anyone is able to predict with an accuracy of around 72.4% on data set that is randomly sampled. Thus, we do not claim that the high prediction accuracy of our classifier for ethnicity comes from the advantage of having access to real user's personalized ads. High bias in sample distribution (*e.g.*, ethnicity) is known to result in biased classifiers. Unfortunately, we find the distributions of our subjects are biased for age, education, income, political affiliation, religion, and marital status as well, due to the small size of our dataset.

To remedy the aforementioned limitation of our data set, we reorganized our data to make it more evenly distributed between different classes (i.e. different values in each studied category). To this end, we merged some of the less popular classes in age, marital

Table 3.2: Accuracy of classifiers of demographic categories.

	Age	Education	Ethnicity	Gender	Income	Marital Status	Parental Status	Political Affiliation	Religion
Decision Tree	**0.51**	0.30	**0.76**	0.64	**0.47**	**0.62**	0.62	**0.50**	0.35
Logistic Regression	0.38	0.37	0.72	0.73	0.45	**0.62**	0.65	**0.50**	0.39
Multinomial NB	0.37	0.35	0.61	0.73	0.36	0.49	0.65	0.41	0.43
K-Nearest Neighbors	0.39	0.34	0.75	0.65	0.45	0.47	0.59	0.45	0.40
Random Forest	0.39	0.36	0.68	0.67	0.43	0.59	0.58	0.46	0.41
SVM	0.49	**0.40**	0.75	**0.74**	**0.47**	0.59	**0.66**	0.49	**0.46**
Dummy	0.20	0.25	0.20	0.50	0.33	0.33	0.50	0.33	0.33

status, political affiliation, income and education; the distributions of demographics in the merged classes are as shown in Table 3.3. As a result, some of the previous multi-class classification problems are reduced to binary classification problems. We augmented the dummy classifier by using a majority selection strategy, *i.e.* it always outputs the most popular label in the training set. Table 3.4 lists the accuracies of all the classification models on the new classification problems. Since the number of classes in the 5 demographic categories was reduced, we observed improvement on the performance of classifiers. The accuracies of our classifiers for age, income, marital status and political affiliation are better than random guess and the case with prior knowledge of population distribution. For instance, we could accurately learn the income, or marital status for more than 60% of users. The information derived from personalized ads indeed helps one predict a users' personal information with better accuracy. However, the price of the performance improvement is the coarser granularity of the prediction. For example, the new classifier can not differentiate people whose annual income is higher than $60K with people who earn less than $60K but higher than $30K per year. None of the classifiers performs significantly better than augmented dummy classifier for education and religion, which suggests the adversary has little advantage for the two categories.

In addition to studying the possibility to predict individual aspects of the user's demographic, we also try to determine for each user how many of the 9 studied demographic categories the adversary may learn by monitoring his or her personalized ads. We record the accurate demographic predictions of each user in the 5-fold cross validation. Figure 3.14 presents the distribution of the number of correct predictions for demographic categories across users. For 91% of the users, at least 4 demographic categories of the users could be

Table 3.3: Reorganized distribution of demographics of subjects.

Age			Marital Status		Political Affiliation	
18-27	28-33	34+	Single	Not single	Independent	Non-Independent
71	71	75	124	93	108	109
32.72%	32.72%	34.56%	57.14%	42.86%	49.77%	50.23%

Income		Education		
$< \$30K$	$> \$30K$	High school or less	Associates	Bachelor or higher
107	110	78	50	89
49.31%	50.69%	35.94%	23.04%	41.02%

Table 3.4: Accuracy of classifiers of reorganized demographic categories.

	Age	Education	Ethnicity	Gender	Income	Marital Status	Parental Status	Political Affiliation	Religion
Decision Tree	0.52	0.38	0.75	0.64	0.54	0.60	0.63	0.54	0.38
Logistic Regression	**0.54**	0.41	0.73	0.72	0.57	0.59	0.64	**0.59**	0.42
Multinomial NB	0.45	0.44	0.73	**0.75**	**0.62**	0.61	0.64	**0.59**	0.41
K-Nearest Neighbors	0.41	0.37	0.74	0.60	0.52	0.62	0.56	0.54	**0.43**
Random Forest	0.42	0.40	0.71	0.70	0.51	0.60	0.60	0.58	0.38
SVM	0.44	**0.45**	**0.75**	0.73	0.54	**0.63**	**0.66**	**0.59**	**0.43**
Dummy	0.33	0.33	0.20	0.50	0.50	0.50	0.50	0.50	0.33
Augmented Dummy	0.35	0.41	0.75	0.56	0.51	0.57	0.59	0.50	0.41

accurately predicted[10]. There are even 2 users that the predictions for all the 9 demographic categories are correct.

Summary of Results. We demonstrated the possibility of leaking user's sensitive personal information through personalized mobile ads to third-party app developers. With data from about 200 users, we were able to build classifiers that predict gender with over 70% accuracy and parental status with over 65% accuracy. By balancing the subject distribution in age, income, political affiliation, and marital status, we could predict a user's corresponding demographic class with significantly better accuracy than random guess and the case with prior knowledge of population distribution. We are not able to build classification models that have significant advantage over an augmented dummy classifier for education, ethnicity and religion. We discuss the privacy implication of our results next.

3.5.4 Interpreting our Results

Our results presented in Table 3.2 and Table 3.4 indicate that our ability to predict a user's gender, age and parental status is significantly higher than that of predicting other types

[10]This can be different set of 4 categories for different users

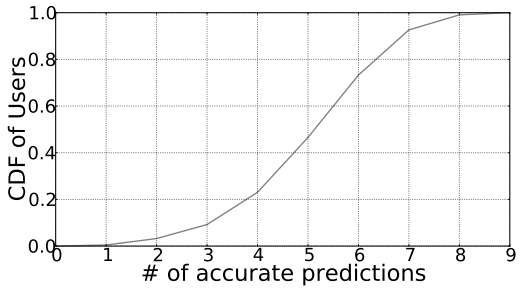


Figure 3.14: Number of accurate predictions for demographic categories across users.

of demographics information. On the surface, this is neither surprising nor alarming in a privacy perspective. It is unsurprising because as seen in [50, 47]; gender, age and parental status are the three targeting options offered in Google's current ad product. Therefore, one can deduce user's demographic information in these categories based on what ads they see. Simply put, *our results confirmed that Google can deliver on what it is offering advertisers; it can correctly deliver ads to the specified demographic groups.* One can also argue that the privacy concern caused by leaking one's gender, age and parental status is very minimal.

However, the real surprise lies in the adversary's non-trivial gain in his/her ability to predict aspects of the user's demographic other than age, gender and parental status. Some of the other demographic information (e.g. political beliefs) is deemed so sensitive that Google explicitly stated [59, 60] that they will not even be collected as part of a user's profile; for the rest, there is no known documentation that suggests Google is using them for personalization. We believe our success in gaining some knowledge of these other aspects of the user's demographic can be explained by the Federal Trade Commission's study on data brokers [61], which shows that general non-sensitive information collectively could be used to infer more sensitive information. In particular, it is very possible that there are very strong correlations between one's age, gender and parental status with his/her other demographic information. For example, parents are much more likely to be married, and older people generally have higher incomes. This highlights the more profound privacy implication of our work.

Privacy Implication. In the traditional web settings, Google can protect its users by en-

forcing published policies that prohibit the collection of the more sensitive user demographic information. This is an effective protection since none of Google's profiling information is leaked to the websites that host advertisements. However, such protection is no longer effective in mobile settings, where any ad-hosting app can observe the personalized advertisement being shown to the user. In this new ad-hosting environment, Google is, to a very large extent, sharing its profile of the user with the app developers and potentially other ad-networks, and Google cannot dictate how this shared/leaked information is used. As shown in our results and [61], even the sharing of the most benign user demographic information can have the adverse effect of allowing third parties to gain some sensitive demographics information about the users, and the threat from the privacy leak through this new channel is only going to increase when Google starts using other "benign" user demographic information for ad personalization.

Due to the lack of separation between in-app advertisements and the rest of the host app logic, the host app can observe all personalized advertisements without needing any extra permission. Since ad personalization is inherently performed based on the personal information of the user, by revealing to the host app what ad is being displayed to the user, the ad network may be inadvertently leaking some of its collected user information to the app developer. Our study shows that such leaked information can be used to accurately derive some of the user's demographic information. This is especially true for information like gender, age, and parental status, which are known to be used in ad targeting. In addition, some information that ad networks might not be explicitly collecting or using could also be leaked to the app developer. Our results indicate that one can predict significantly better a user's income, political affiliation, and marital status over random guess by observing the personalized ads that are served to a user. The information thus inferred may then be used to request ads from other higher-paying ad networks. The leaked user information may also be used for price discrimination. For example, the same good could be sold at different prices to users in different income groups. Furthermore, the private information can also be sold or

transferred to other parties.

Our results highlight the need for protecting private user data (personalized ads) from unauthorized parties (app developers) on Android. The equivalence of Same Origin Policy should be provided on Android to isolate personalized ads from application context to protect user's privacy. Proposals like AdSplit [48] and AdDroid [62] are a good starting point for separating ads from applications on Android. Furthermore, before such isolation between in-app advertisement and the host app is established, ad networks should balance the gain in revenue and the risk of the user's privacy when they decide to personalize ads using more detailed or sensitive user demographic information.

3.6 Discussion

3.6.1 Limitations

The main limitation of our study is the small sample size and the uneven demographic distribution of our data set. We argue that such limitations do not invalidate our results. The ads that are correlated with demographics are selected by applying statistical tests. And since we are using a significance level of 0.005 in our statistical test, we are 99.5% confident that the correlations are not observed by chance. Similarly, by using 5-fold cross validation for evaluating our ability to learn user's demographic information based on the ads he/she receives, our results in Section 3.5 confirms that the threat of leaking sensitive user information through personalized ad is real. Furthermore, we argue that aggressive/malicious app developers or ad-networks can achieve significantly better accuracy than what we've shown in Section 3.5 for two reasons: 1) they can invest more resources to obtain better ground truth data, and 2) they can observe ads received by users for a longer period of time (and thus have more highly personalized ads in their data set). In future work, we plan to apply our technique to other ad-networks and to attempt to collect ad data for a longer period of time. We will also try to improve our results in Section 3.5 by experimenting with techniques to better clean our data set (e.g. remove users who appear to receive a lot of

non-personalized ads) as well as techniques like multi-task learning to better leverage our advantage in predicting the user's gender, age and parental status.

3.6.2 Countermeasures

The root cause of the studied privacy leak from personalized ads to the hosting application is the lack of isolation between the ads and the app. Thus, adopting HTTPS to protect the ad traffic will not stop the problem. While previous work [48, 62] highlighted the need for isolating ad libraries largely from the perspective of separating permissions of ad-related code from code of the hosting app, our work in Section 3.5 shows that there is also a need to prevent the hosting app from reading the ad library's data when that data is derived from the ad-network's private information.

As the essential core of the mobile advertising ecosystem, ad networks are responsible for protecting users' privacy. Since the above ad isolation techniques have not been widely adopted, ad providers should build defense mechanisms into their products to protect users' privacy. One possible defense could be adding noise or randomness into the personalized results. For example, ad networks could make a larger fraction of their ad deliveries to be non-personalized or contextual ads instead of maximizing personalization of every ad impression. A similar technique has been proposed for the scope of privacy protection in online searches (*e.g.* adding noise into user's search history) [63], and could make it more difficult for an adversary to learn user's personal information and mitigate part (if not all) of the privacy threat we identified in Section 3.5.

Besides adding noise to personalized ads, ad networks may also provide coarser grained targeting options for advertisers. For example, instead of enabling advertisers to precisely target users that are 26 years old, ad networks may only provide a range (*e.g.* 25-34) for targeting. Such approaches may result in coarser granularity of adversary-accessible personal information and decrease the severity of privacy leakage. Google AdMob is already offering ad targeting only for coarse-grained age groups; we encourage other ad networks to

adopt a similar model in their targeting offerings.

The idea behind both of the proposed countermeasures is to trade off the quality of ad personalization to limit the degree of privacy leak through such ads. We cannot expect all ad networks to adopt such an approach because less personalized ads may contribute to a loss in ad revenue. We will leave it as an open problem to identify a strategy that can avoid such trade-off and still work in the current ad-hosting environment (where there is no isolation between the logic/data of the ad-library and the main app).

3.7 Related Work

Privacy in Online Advertising. The privacy issues related to online advertising have been the focus of quite a number of studies. For example, Roesner *et al.* [27] showed the prevalence of third-party web tracking and designed a browser extension for defending against social widget tracking. Acar *et al.* [64] studied three advanced web tracking mechanisms - canvas fingerprinting, evercookies and the use of "cookie syncing" in conjunction with evercookies - and suggested that even sophisticated users can face great difficulties in evading tracking. XRay [65] tried to identify how various tracking information is being utilized by targeted ads. Korolova [66] presented attacks that exploit Facebook's advertising system to infer private user information. Barford *et al.*[67] found widespread use of ad targeting mechanisms on the web and showed significant correlation between user interest profile and in-browser ads. Olejnik *et al.* [41] examined the leakage of users' browsing histories through Cookie Matching and Real-Time Bidding. Datta *et al.* [68] explored how user behaviors, Google's ads and Ad Setting interact in the web settings. Finally, Castelluccia *et al.* [40] demonstrated an attack very similar to what we presented in Section 3.5, where the adversary tries to reconstruct user's interest profile from unencrypted personalized in-browser ad traffic of synthesized users.

These works focus mainly on advertising in web pages. In contrast, our work focused on similar issues in an in-app advertising setting. As we have mentioned in the introduction, not

only is the personalization of in-app advertisements less understood, it also has the potential to raise more serious privacy concerns, due to the intimate nature of mobile phones. Also, as compared to [40] which requires the adversary to have the capability to intercept the victim's network traffic, the attack we presented in Section 3.5 can be carried out by any app on Android.

Privacy-Preserving Advertising. A number of systems have been proposed for privacy-preserving personalization. Privad [69, 70], Adnostic [71], and RePriv [72] achieved this goal by using generalized user profiles and moving ad personalization to the client side. ObliviAd [73] leveraged secure hardware to provide privacy guarantees. Mor *et al.* [74] designed Bloom cookies for encoding a user's profile in a privacy-preserving manner. Hardt *et al.* [75] proposed a differentially private distributed protocol that simultaneously achieves reasonable level of privacy, efficiency and quality in personalization on smart phones. While these proposals protect users' private information or identifiers from being leaked to ad networks, they cannot stop the attack in Section 3.5, which only requires observation of the end results of personalization.

Mobile Ad Personalization. Nath [38] presented MAdScope for harvesting in-app ads and characterizing in-app targeted ads. By studying the keywords/ad-control attributes included in ad requests from different apps, the author found that only one of the top ten in-app ad networks is using behavioral targeting, and demographic information is not commonly used in in-app ads. Book *et al.* [43, 39] surveyed how app developers used ad control APIs to show ads targeting their presumed user population, and studied mobile ad targeting using simulated user profiles and found targeting based on users. In contrast to these two studies, we focused on personalization in the absence of any input from the app developers, and instead of using synthesized user profiles, we harvested ads and demographic information from real users. Our results suggested that a large fraction of ad impressions are correlated with demographic information.

Ad Isolation on Android. Recent studies on isolating advertising from application could

provide solutions to the privacy leakage problem we studied in Section 3.5. AdSplit [48] is an Android extension that allows an app and the ad library to run as separate processes with different permissions. AdDroid [62] also separates privileged advertising functionality from host applications on Android. Roesner *et al.* [76] designed LayerCake to support cross-principal embedded interfaces on Android.

3.8 Summary

In this chapter, we have studied how user information is utilized by major ad providers for in-app ad personalization on mobile devices and to what extent ad networks know about the user's interest and demographic information. We have also investigated if in-app advertisements can be a channel for leaking user information collected by ad networks to apps hosting these advertisements. We demonstrated that a malicious app can indirectly infer potentially sensitive personal information by hosting third-party personalized in-app ads. Specifically, we achieved high accuracy in demographic categories that are explicitly used as targeting options, and showed that information that is not used in serving tailored ads could also be inferred by app developers. These findings illustrate that more protection is needed to protect indirect user data from malicious first-party mobile applications.

CHAPTER 4
SELECTIVE CONTROL ON WEB TRACKING

4.1 Motivation

Recent advance in online tracking technologies are putting an unprecedented amount of user information into online vendors' hands. These information are surprisingly vast, from search queries to web browsing behavior to purchase history and more – and all together, they can be used to predict user preference. As such, online tracking brings many privacy concerns [66, 40].

Unarguably, online vendors are generous investors and untiring advocates of tracking techniques. Using tracking techniques to acquire more information about users, online vendors can improve their conversion rates and enable their business partners to be more economical with their advertising and marketing budgets. However, investments in tracking techniques can be severely undermined if users disable tracking and refuse to share any of their online footprints with vendors due to privacy concerns. Indeed, we have already seen a significant growth in the adoption of anti-tracking tools and software [77].

To combat users' fear of privacy invasion, recent marketing research [78, 79] provides online vendors with many suggestions, such as being transparent and simple about privacy policies, building users' trust in personal information use, and giving users options to manage the use of their shared information. Presumably, in response to these suggestions, online vendors take active measures. For example, Uber summarizes their privacy policies in an easy-to-read bullet format that does not make users' eyes glaze over [80]. Both Google and Yahoo provide tools that allow users to manage their privacy sensitive data [81, 82].

While vendors taking care of privacy concerns build users' trust and increase users' willingness to share information, there are still a significant amount of users who actively limit involuntarily sharing of data. The reason is vendor-provided tools do not prevent

them from logging a user's privacy sensitive visits but rather restricts how they use these data, meaning that the protection of such user's privacy is completely relying on vendors. Therefore, many users concern if vendors truly have proper mechanisms in place to protect their privacy sensitive information [83, 84, 85].

An intuitive solution to this conundrum is to provide users with a client side tool that shares the same functions with the tool provided by online vendors. Different from the vendor's tool, however, the client side solution provides users with a power, that is, only to share information that they are comfortable with but not disclose their privacy sensitive visits. In achieving this, this chapter proposes `TrackMeOrNot`, a novel anti-tracking mechanism that prevents online vendors from tracking privacy sensitive visits specified by users. More specifically, we augment conventional web browsers with the ability to take a user's privacy demand, and shield her browsing activities based on her need.

Existing client side anti-tracking mechanisms completely impede vendors' tracking and vendors cannot obtain any information from users. Considering that online vendors also use user data to offer personalized online experience, these solutions can severely disrupt user experience. To this end, the goal of `TrackMeOrNot` is to allow users to trade the information that they are comfortable to share for a better user experience. In order to achieve such a goal, `TrackMeOrNot` has to address following two unprecedented challenges. First, `TrackMeOrNot` needs to correctly understand the semantic meaning of a web page to determine if the visit violates user's privacy demand (positive). `TrackMeOrNot` may leak many privacy sensitive page visits to vendors with a high false negative rate. On the other side, `TrackMeOrNot` may negatively affect user's browsing experience if lots of false positives are triggered. Our first challenge is how to build an efficient scheme that accurately determines if disclosing a visit to vendors' trackers violates a user's privacy demand. Second, to avoid exposing privacy sensitive page visits to vendors' trackers, `TrackMeOrNot` needs to examine if a page visit violates the user's privacy demand in advance of loading the page into a web browser. Browser interception for protecting

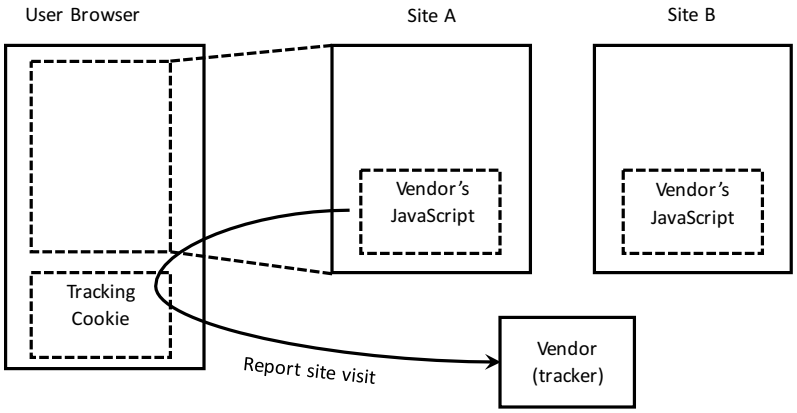


Figure 4.1: Online tracking workflow.

privacy can introduce unexpected overhead, and sluggish examination can decelerate page loading and jeopardize user experience. As a result, our second challenge is how to build a lightweight interception scheme for privacy protection.

In this chapter, we address the aforementioned challenges as follows. First, we introduce an efficient approach to examine a user's visit and her privacy demand. This approach decouples web page content from that of tracking parties, and excludes the unnecessary page rendering. Second, we introduce a lightweight browser interception scheme. It reduces memory consumption by maintaining only one additional browsing context for each browser tab. These two contexts allow a web browser to quickly and smoothly cloak a user's browsing session accordingly.

4.2 Background

An online vendor typically partners with many websites. Using a JavaScript snippet embedded on the partner sites, it places a persistent identifier (*e.g.*, tracking cookie) on a user's browser. Every time a user visits a vendor's partner site, the JavaScript snippet reports the visit to the vendor along with the persistent identifier associated with the user's browser. As such, the vendor can link a user's browsing activities across multiple sites, build the user profile and tailor his or her browsing experience. Figure 4.1 illustrates this tracking process.

Considering users may not be comfortable with disclosing all her footprints to vendors,

many vendors provide web portals that allow users to opt out their unwanted footprints. The opt-out option shows vendors' respect on user privacy and have been accepted by many users. However, there are still a significant number of users who are concerned that vendors may not be able to properly protect their privacy footprints because the opt-out only restricts vendors to use unwanted data rather than completely preventing vendors from collecting unwanted data.

Existing client side anti-tracking mechanisms (*e.g.*, disabling third-party cookie or blocking tracking traffic) could completely impede vendors' tracking such that the trackers cannot collect any information from users. While providing strong protection for users' privacy, such mechanisms jeopardize user experience because – in addition to yielding profits – vendors cannot tailor user's online experiences anymore using his or her footprints in the past.

To the best of our knowledge, none of the existing controls on web tracking could protect users' privacy while preserving usability. We argue that privacy and usability do not have to be mutually exclusive and observe the need for new anti-tracking mechanisms that could balance between users' demands for privacy and usability. Such anti-tracking mechanisms should meet the following requirements: 1) user's privacy sensitive browsing activities should not be known to any vendors; 2) vendors should be able to collect data about browsing activities that have been explicitly granted by the user. In addition, any anti-tracking mechanisms should not negatively affect user's browsing experience (*e.g.*, introducing acceptable overheads such as latency, computation and memory usage).

In this work, we propose a new anti-tracking mechanism that allows users to selectively share their browsing activities with online trackers for better user experience while shielding their sensitive browsing activities. We emphasize that our goal is not only to develop algorithms that determine if a visit violates a user-specified privacy need, but also to develop a system that can effectively and efficiently decouple tracking identifiers from his or her privacy sensitive visits. Our proposed anti-tracking mechanism is mainly used for defending

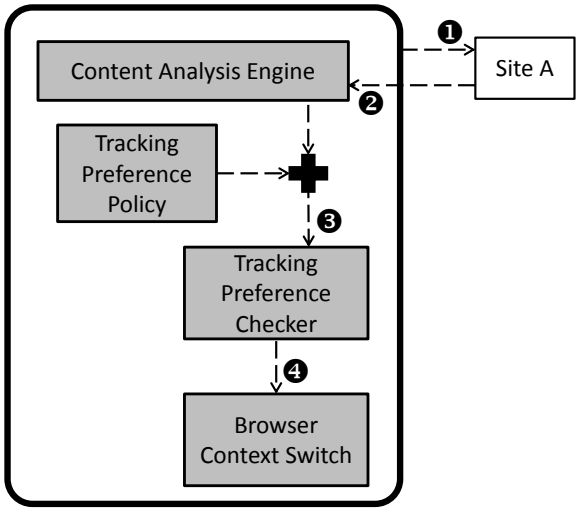


Figure 4.2: The overall workflow of `TrackMeOrNot`.

against stateful tracking techniques [26], such as HTTP cookies and supercookies. But, it can also be further extended with stateless tracking defense techniques, e.g., avoiding browser fingerprinting [86].

4.3 Design

In this section, we present our design of `TrackMeOrNot` to support fine-grained content-aware control on first-party and third-party web tracking.

4.3.1 Overview

As is described in §4.2, users have different and unique tracking privacy needs to online web tracking. To this end, we design `TrackMeOrNot` and depict its overall workflow in Figure 4.2. In order to balance between such privacy needs and usability, `TrackMeOrNot` employs a tracking preference policy, in which users can easily specify their own privacy needs regarding online tracking (§4.3.2). `TrackMeOrNot` separates user's sensitive browsing activities from user's normal browsing activities by leveraging isolated browsing context, which consists of all local states (*e.g.*, cache, cookie jar, local storage, *etc.*) associated with one browser instance. All navigation in `TrackMeOrNot` starts with an anonymous and transient browsing context. During the web navigation, `TrackMeOrNot` analyzes the

semantic meaning of the target web contents (§4.3.3) to interpolate its analysis results with a privacy policy. Then TrackMeOrNot leverages such learned information to determine which browsing context (persistent or anonymous) it should be running to meet the user's tracking preference policy (§4.3.4). Finally, TrackMeOrNot carries out seamless browsing context switch based on the switching decision (§4.3.5).

4.3.2 Tracking Preference Policy

Following the general and well-known beliefs in user's privacy protection, TrackMeOrNot allows users to specify their privacy needs on their own. These privacy needs in general can be expressed with the well known access control concepts, blacklists and whitelists. Leveraging these basic schemes, TrackMeOrNot introduces two different types of protection entities, web content category and domain name. Web content category offers the capability to specify user's privacy needs from a bag of category words. For example, a user can enlist the category *drug* into the blacklist of the web content category, if the user wants to hide the fact that she or he has browsed a web page related to drugs. On the other hand, *drug* can be whitelisted if the user does not feel those contents are privacy sensitive and thus wants to receive useful services based on those contents (e.g., targeted advertisements from third parties). Moreover, the other protection entity, domain name, allows users to easily specify their privacy needs using the domain name itself. For example, a user can whitelist *cnn.com*, if she/he believes the contents offered by *cnn.com* are not privacy sensitive and thus allow all third-party trackers loaded on *cnn.com* to trace her/his visits.

Similar to access control systems, TrackMeOrNot also defines override rules on privacy policies. A user could assign a high priority to a whitelist rule so that TrackMeOrNot disregards blacklist rules with lower priorities if the whitelist rule is ever matched. If two rules share the same priority, a blacklist rule always precedes a whitelist rule. TrackMeOrNot supports the following three custom priorities for each policy rule that users can specify: *high*, *medium*, and *low*, where *high* and *low* indicate the highest and lowest priority respec-

```
{
    "category-blacklist": {
        "drugs": "high"
    },
    "category-whitelist": {
        "sports": "low"
    },
    "domain-blacklist": {
        "aaa.com": "medium"
    },
    "domain-whitelist": {
        "bbb.com": "medium"
    },
    "fallback browsing mode": {
        "anonymous"
    }
}
```

Figure 4.3: Tracking preference policy for `TrackMeOrNot`.

tively. For example, suppose the user specified *drug* as a category blacklist rule with a *medium* priority and *cnn.com* as a domain whitelist rule with *high* priority, `TrackMeOrNot` will first apply the policy on *cnn.com* and then disregard the policy on *drug* if the policy on *cnn.com* is matched. By default, all rules have the same priority (medium) if the user does not define explicitly.

Furthermore, users can specify a fallback browsing mode, which specifies which browsing context would be used in case none of the policies are matched. A privacy conscious user may select *anonymous* as her or his fallback browsing mode and specify only a few web content categories and domains as whitelist rules. On the other side, a user who is more concerned with usability (*e.g.*, the quality of personalization service) may use *normal* as her or his fallback browsing mode and enlist those categories and domains that she or he determines as sensitive into blacklist. We note that how a user defines her or his tracking preference policy impacts the performance of `TrackMeOrNot` as we will discuss in §4.6.

Figure 4.3 depicts an example of the privacy preference policy that `TrackMeOrNot` accepts. Each protection entry (i.e., either category or domain) has both blacklist and whitelist policies. In each policy, the protection entity and priority are specified as a key/value pair. Moreover, a `fallback browsing mode` is specified as `anonymous` in this example, meaning that the anonymous browsing context would be used as the fallback mode.

4.3.3 Content Analysis Engine

In order to determine whether a target website that a user is visiting is against any blacklist policies, TrackMeOrNot intercepts web navigation processes to analyze web contents that the browser receives and renders. In particular, TrackMeOrNot first intercepts all the network responses from the first party, and then it starts the content analysis with the built-in machine learning classifiers to obtain the representative category of the responses.

TrackMeOrNot collects all the network responses by intercepting the browser's event on completing the fetching of a resource. TrackMeOrNot particularly focuses on collecting the responses only from the first party for the following reasons. First of all, most of the content semantics are delivered by the first party (i.e., the third party mostly delivers external data including images, tracking scripts, or advertisements), and these semantics are what users are concerned about from the privacy point of view. Furthermore, it would require too much analysis time if TrackMeOrNot also considers the resources from the third party as well because the loading of third party resources can take quite long time. Again, the resources from the third party are mostly related to external data, which usually have a significantly bigger size than those from the first party.

Once such an event is fired, TrackMeOrNot forwards the corresponding response contents to a content analysis engine. The analysis engine first parses responses using its own HTML parser. Since the HTML parser provided by the underlying browser aims at rendering the complete web page from the responses, it is not designed to parse incomplete responses. Thus, TrackMeOrNot implements a custom HTML parser, which is capable of handling such incomplete responses and focuses on extracting text semantics. Specifically, this parser constructs the DOM tree without sending new network requests nor evaluating JavaScript and Cascading Style Sheet, each of which is not related to the content semantics. Next, TrackMeOrNot scans the DOM tree and extracts texts from it. Non-content texts (i.e., navigation links and advertisements) are excluded from the final result as those are not related to the real content semantics.

TrackMeOrNot then utilizes pre-built machine learning classifiers to summarize the extracted text contents. For example, a classifier may predict the likelihood that the current web page contains pornography content. Another classifier may predict whether the web page is about education. The summarized classification results (probabilities) are then sent to the central controller for checking with user tracking preference, which we describe in the next subsection. If the built-in machine learning classifiers are not satisfactory to some users, they could also select some free online classification services for processing their navigation request before the navigation request is sent in TrackMeOrNot. The correct browsing context could be directly decided after checking the returned results with user privacy preference. However, the users have to build their privacy protection on trust of the third party service providers and may suffer from unpredictable browsing latency.

While the content analysis engine is processing the content semantics, the page loading process is not blocked so that TrackMeOrNot introduces minimum overhead in page load time if TrackMeOrNot does not switch the browsing context.

4.3.4 Tracking Preference Checker

Once the content analysis is finished, TrackMeOrNot checks the privacy sensitivity of the target website. In other words, using the tracking preference policies and the target website's content semantics, TrackMeOrNot determines whether it needs to keep using anonymous browsing context to prevent tracking or to switch to persistent browsing context to augment usability.

As described in §4.3.2, TrackMeOrNot checks each policy rule in the order of associated priority. For example, suppose a user specified *drugs* in category blacklist as depicted in Figure 4.3. And further suppose that the content analysis engine returned the target website has the semantics, drugs. In this case, TrackMeOrNot determines that the browsing context does not need to be changed. However, if drugs were specified in category whitelist, TrackMeOrNot will switch to the persistent browsing context, which we describe in detail

in the next section.

4.3.5 Browsing Context Switch

Based on the current browsing context and the decision made after checking user's tracking preference, `TrackMeOrNot` may switch to the persistent browsing context for the current navigation request. If a browsing context switch is not needed, `TrackMeOrNot` simply does nothing, meaning that the browsing context would be stayed in the anonymous one. Otherwise, `TrackMeOrNot` terminates the current pending navigation originated from the initial anonymous browsing context, and restarts the navigation with the persistent browsing context.

`TrackMeOrNot` will also mark the new navigation request so that it will not be intercepted again. Since the machine learning classifiers cannot be 100% accurate, sometimes `TrackMeOrNot` may switch to a browsing context that the user does not want. In this case, the user could manually switch back to the correct browsing context.

4.3.6 Discussion

By isolating privacy sensitive browsing activities in anonymous browsing contexts, `TrackMeOrNot` effectively prevents online vendors from linking those activities with a user's persistent profile. However, a user may still feel unsafe because other non-local state features may be used to track the user's browsing activities as well. For example, stateless fingerprinting does not use any local states of a browsing context to identify a browser instance. Such stateless tracking threat has already been addressed in [87]. The solutions could be integrated with `TrackMeOrNot` by generating a temporary fingerprint when using the temporary browsing context. IP anonymity technologies [88] could similarly be integrated with `TrackMeOrNot` to defend against IP address based tracking.

Another limitation of `TrackMeOrNot` is that in an anonymous browsing context `TrackMeOrNot` is not able to extract the content of a few websites that only provide services after the user

has logged in. For example, Facebook does not provide its service if the user does not sign in with her or his account. As a result, the user has to explicitly specify a whitelist rule for Facebook if the user wants to use its service, or the user needs to log in her/his account of Facebook in the anonymous browsing context as well. The current design of `TrackMeOrNot` may also allow user's browsing activities on websites like Facebook to be tracked by other third-party trackers, if Facebook and other websites explicitly embed tracking scripts of other vendors into their own websites. However, such behavior is essentially the same as directly sharing first party data with third party trackers, which cannot be prevented if the user has agreed with the terms and conditions of these websites that explicitly indicate that user data will be shared with third-parties.

4.4 Implementation

To demonstrate the feasibility and effectiveness of `TrackMeOrNot`, we implemented a prototype of `TrackMeOrNot` based on one of the most popular modern browsers, Chromium (version 45.0.2426.3). Although we only implemented a prototype on Chromium, the design of `TrackMeOrNot` is generic and can be easily extended to other browser platforms. In terms of implementation complexity, the Chromium version of `TrackMeOrNot` introduced 1,400 new lines of code (200 LoC for navigation interception, 700 LoC for content analysis engine, 500 LoC for tracking preference and browsing context switching) to Chromium.

In the rest of this section, we first introduce the general architectural design of the Chromium browser, and then describe how each component of `TrackMeOrNot` is implemented along with Chromium's design.

4.4.1 Chromium Browser's Architecture

Chromium uses a multi-process architecture [89] to utilize process-level isolation between the browser process and the renderer process. The browser process of Chromium is the main process that manages UI, I/O, tabs, configuration, etc. The renderer process renders the web

page using the WebKit layout engine, and multiple renderer processes are associated with and controlled by the browser process. At the time of creation, the browser process and renderer processes are strictly bound with a certain browsing context (either persistent or temporary browsing context), and the current design does not allow to switch between them after being created.

4.4.2 Tracking Preference Policy

`TrackMeOrNot`'s tracking preference policy leverages existing preference system in Chromium [90], which is managed by the browser process. While `TrackMeOrNot` internally reuses Chromium's preference system, it also provides an interface for users to easily configure her/his tracking preference through Chromium's browser setting interface itself (*i.e.*, we added the `Tracking` section into the `Content settings` option of Chromium's `Privacy` setting). This tracking preference is loaded into a browser process when the Chromium browser is launched, and will be used later to enforce tracking policy based on the content analysis results.

4.4.3 Content Analysis Engine

The content analysis engine is implemented inside WebKit in the renderer process. We intercept each navigation request inside the WebKit layout engine. In our current design, we only intercept network responses of first-party contents in the ResourceFetcher, as is discussed in §4.3.3. The network requests for third-party contents are temporarily held inside the DOM parser of WebKit when a new navigation starts. The hold is released when the renderer receives a notification from the browser process, or is terminated with the navigation request if the browser process switches to a different renderer process with a different browsing context. If a new renderer process is selected for restarting the navigation, the browser process will set a flag in that renderer process so that the new navigation request will not be intercepted again.

In order to understand the semantic meaning of a web page, `TrackMeOrNot` needs to

parse the corresponding HTML source and extract the content from it. However, the DOM parser of WebKit might be blocked by network requests that are held [91], so that we are not able to extract all the text contents with the DOM parser. To overcome this problem, we implemented a lightweight DOM parser with the *libxml2* library [92]. Our DOM parser creates the DOM tree from the HTML source without loading new resources and evaluating JavaScript and CSS. After the DOM tree is built, we extract all the text on the web page. However, the extracted text may contain many non-content text, such as navigation links, copyright disclaimer, or advertisements. Such non-content text may introduce noise in the classification procedure. We implemented the CETD algorithm [93] inside WebKit to further remove non-content text.

A key feature of `TrackMeOrNot` is to use machine learning algorithms for content analysis. While there are many online classification web services, we decided to implement local classification models for privacy and performance concerns that have been discussed in §4.3. Due to the diversity of web pages on the Internet today, we had to train the machine learning models with a large set of diverse web pages. However, to the best of our knowledge none of the widely used web page classification benchmark data sets [94] could meet our requirement because they either contain very few documents that are not diverse enough or use coarse-grained labels. We eventually selected the well known AOL query logs [95] for training our classification models. The AOL query logs include 21 million search queries from 650k real users. The logs also contain over 19 million user click-through events of 1.6 million unique web pages, which had been visited by real users and represent diverse human interests. Some sensitive contents that people generally do not want to be tracked (*e.g.*, pornography) are also included in the data set.

Although the AOL data set is a good fit for our study, unlike other web classification data sets the AOL data set is not labeled. It is impractical to manually label all the URLs in the data set. We used the natural language processing API of AlchemyAPI [96] to label the English web pages that were still accessible in September 2015. Each web page was

pre-processed with the CETD algorithm to remove non-content text before calling the API. AlchemyAPI classifies each web page into topic category up to five levels deep [97]. In total, we were able to label 369,179 web pages with the support from AlchemyAPI [1].

Since some categories have much more web pages compared with other categories, we further broke down web pages in these popular categories into their second level category. For example, "art and entertainment" web pages were further broken into "books and literature", "movies and TV", "music", "shows and events", "visual art and design" and "arts/other". We also manually selected all (second level) categories that are generally considered as sensitive, *i.e.*, "finance", "health and fitness/{disease, disorders, drugs}", "law, govt and politics/{armed forces, government, law enforcement, legal issues}", "society/{crime, dating, sex}". Eventually, we got 78 categories in our data set. This labeled AOL data set was also used in our evaluation of `TrackMeOrNot` in §4.6.

For each of the 78 categories, we built a binary classification model using Linear SVC. The term frequency-inverse document frequency (tf-idf) of each term in the extracted text is used as feature. All the web pages in a given category are positive samples. Equivalent number of randomly selected web pages of other categories are negative samples. The positive samples and negative samples were randomly and evenly split into a training set and a test set. We used 10-fold cross validation in the training set to learn the best parameters for each category, and used the test set for evaluation. The distribution of the ROC AUC scores of the 78 categories are shown in Figure 4.4. The minimum, mean, maximum, and standard deviation of ROC AUC scores are 0.84, 0.92, 0.99, 0.03, respectively.

The prediction methods of the trained models are implemented inside WebKit. For each navigation request, the extracted text is evaluated with all the classification models. The prediction confidence (decision function in the case of LinearSVC) of each binary classifier is gathered to select the best label(s) for the web page. In our implementation, the label of the classification model that predicts positive class with highest confidence is assigned to the

[1] We would like to thank AlchemyAPI for granting us special academic license in this study.

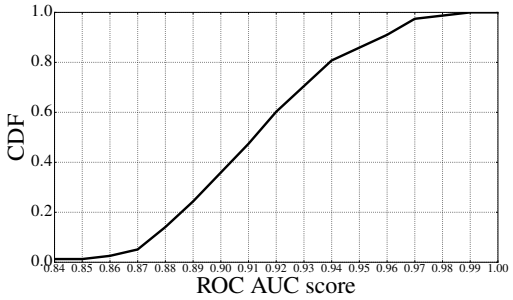


Figure 4.4: The ROC AUC score distribution of binary classifiers.

web page. The final prediction result is then sent back to the browser process for tracking preference check. Since the topic of web pages is very subjective, we plan to enable the users to use their customized classifiers for their own need in the future.

4.4.4 Tracking Preference Checker

Given the content analysis result returned from the renderer process and the user specified tracking preference, this phase determines whether the current navigation leads to content that is privacy sensitive to the user. Based on the result, `TrackMeOrNot` decides whether to switch the browsing context or not. This decision procedural follows the algorithm as illustrated in §4.3.2. If `TrackMeOrNot` determines that the browsing context should not be switched (i.e., keep using the anonymous browsing context), `TrackMeOrNot` notifies the corresponding renderer process to continue rendering the current web page and does not perform any additional operations because the renderer process is already handling the navigation using the anonymous browsing context. On the other hand, if it has to switch to the persistent browsing context, then `TrackMeOrNot` will terminate the current navigation request that is being processed in the renderer process and then switch the browsing context as we describe more details in the next subsection.

4.4.5 Seamless Browsing Context Switch

In Chromium, one browser process is bound with one fixed browsing context. As a result, tabs of different browsing contexts (users) cannot be merged within one browser window.

A naive implementation of TrackMeOrNot will pop-up a new browser window each time the browsing context is switched, which is very annoying to users. To preserve good user experience, we break the binding between one browsing context and one browser process in Chromium. TrackMeOrNot allows one browser process to be associated with multiple browsing contexts so that the browser process could create and manage renderer processes with different browsing contexts. All browsing context switching in our implementation are seamlessly done in the same browser tab without annoying the users. To make the browsing context switching transparent to the users, we also switch the theme of the browser UI when switching the browsing context so that the user could easily tell which browsing context she/he is browsing with. If the user ever wants to use a different browsing context for a navigation, she/he could click a "switch" button on the navigation bar to manually switch the browsing context.

4.5 System Performance Evaluation

In this section, we present the system performance evaluation of TrackMeOrNot. A vanilla build of Chromium (version 45.0.2426.3) was used as the baseline system for comparison. We measured the web page load time and peak memory usage of the two browsers to show the impact of TrackMeOrNot on browser performance and user experience. All experiments were run on Debian Jessie (Linux Kernel 3.16) with a quad-core 3.20 GHz CPU (Intel Xeon W3565) and 24 GB RAM.

We selected the Alexa top 100 US websites as the data set for system performance evaluation. Specifically, the two browsers (i.e., the vanilla Chromium and TrackMeOrNot) sequentially visited the main page of the top 100 websites three times. Note that some top websites (*e.g.*, googleusercontent.com) can not be directly visited from the browser. Thus, we selected the next top websites to fill the places of such websites. We report the average of the three measurements in all the following results.

Depending on the user tracking preference and the web page the user is navigating to,

TrackMeOrNot can exhibit two different browsing behaviors: 1) staying in anonymous browsing context; and 2) switching to persistent browsing context. Intuitively, switching to persistent browsing context in the navigation would impose more run-time overheads in terms of both page load time and memory usage. To clearly understand how much more overheads are imposed in TrackMeOrNot, we configured two user tracking preferences for each of the web page to instruct TrackMeOrNot to either stay in the current browsing context (*Content Analysis Only Configuration*, or *CAOC*) or switch to a different browsing context (*Browsing Context Switching Configuration*, or *BCSC*) in the three visits.

4.5.1 Page load time

Regardless of user tracking preferences, TrackMeOrNot has to always perform content analysis and tracking preference check, which would result in navigation latency. Additionally, TrackMeOrNot may reload the web page using a different browsing context based on the result of tracking preference check that further extends the page load time. In order to precisely measure such extra latency, we first implemented internal hooks in various critical event handlers in Chromium (*e.g.*, page load completion event), each of which measures a clock in nano-second precision using `clock_gettime()`.

Figure 4.5 and Figure 4.6 present the result of main page navigation in case of CAOC and BCSC, respectively. The page load time overhead is the ratio of the extra load time introduced by TrackMeOrNot to the complete page load time using the vanilla Chromium.

When configured with CAOC, TrackMeOrNot introduced negligible extra latency - 0% to 10% overhead in the page load time with 1.93% as mean and 1.81% as standard deviation. We believe the content analysis engine of TrackMeOrNot is very efficient and would not disrupt the user's navigation experience in practice for the CAOC case, because minimum, average, maximum and standard deviation of extra processing time were only 1.00 ms, 39.56 ms, 232.00 ms and 37.40 ms, respectively.

When configured with BCSC, TrackMeOrNot needs to restart the navigation with the

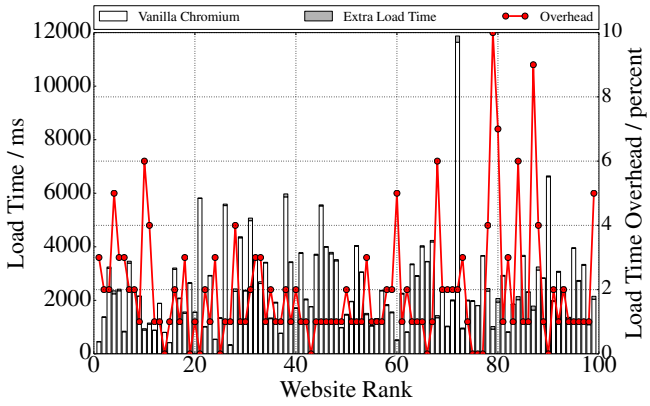


Figure 4.5: Page load time when visiting each of Alexa top 100 US websites under Content Analysis Only Configuration (CAOC).

persistent browsing context, thereby incurring more latency to page load time. From our evaluation, the overhead varied significantly: the minimum, average, median, maximum and standard deviation of load time overhead were 1.00%, 16.40%, 13.00%, 54.00% and 12.92%, respectively. The extra load time (minimum: 51.00 ms, mean: 340.33 ms, median: 228.00 ms, maximum: 2130.00 ms, standard deviation: 352.60 ms) is inconsistent with the overhead, because the raw page load time itself using the vanilla Chromium varied a lot. For example, the vanilla browser loaded www.google.com using 451 ms where `TrackMeOrNot` spent 198 ms on resending the request to www.google.com (including content analysis), imposing 44% (198/451) overhead in page load time. On the other hand, `TrackMeOrNot` only took 148 ms to switch browsing context for www.nytimes.com where the complete page of www.nytimes.com needed 5,539 ms to load using vanilla browser, resulting in only 3% (148/5539) overhead in page load time. Considering that half of the main pages of US top 100 websites needed more than 2,145 ms (median) to load using vanilla Chromium, we believe the delay in page loading (median: 228 ms) caused by `TrackMeOrNot` would not be observable to normal users.

To better understand the latency introduced when using BCSC, we also recorded the time to load the main frame HTML source using the vanilla Chromium for each web page. We present the side-by-side comparison of main frame HTML source load time of vanilla Chromium and extra page load time of `TrackMeOrNot` in Figure 4.7. As is evident from

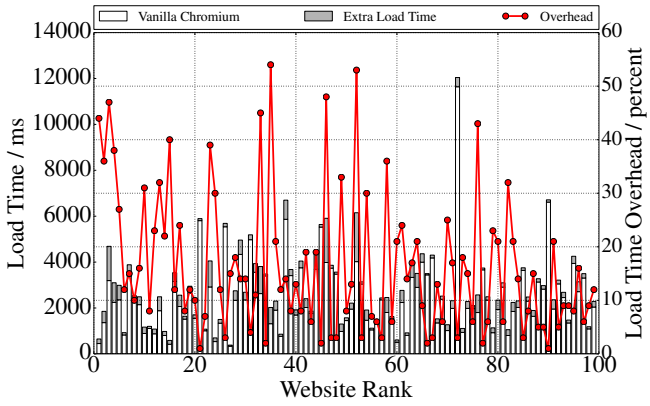


Figure 4.6: Page load time when visiting each of Alexa top 100 US websites under Browsing Context Switching Configuration (BCSC).

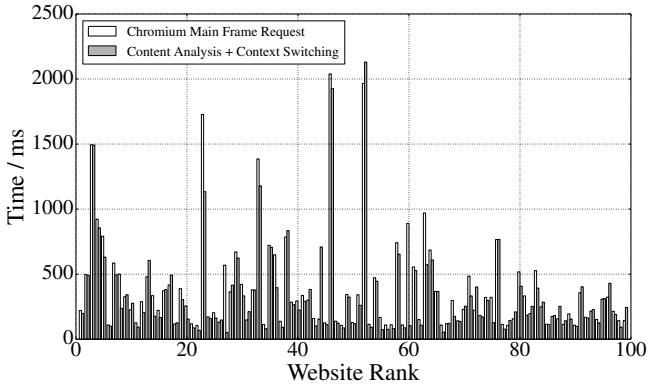


Figure 4.7: Main frame HTML source load time (marked as white boxes) *v.s.* extra page load time in `TrackMeOrNot` (marked as black boxes) when visiting each of Alexa top 100 US websites.

the figure, the extra page load time closely matches with the time needed for loading the HTML source of main frame. If the full page load time is not significantly higher than the main frame HTML source load time, then `TrackMeOrNot` will lead to very high overhead in page load time. However, `TrackMeOrNot` could be enhanced by caching the main frame HTML source that is loaded in the previous browsing context and sharing the cached HTML source with the new renderer process. Thus no extra request needs to be sent, which could significantly reduce overhead in page load time introduced by `TrackMeOrNot`. We leave this implementation optimization as our future work.

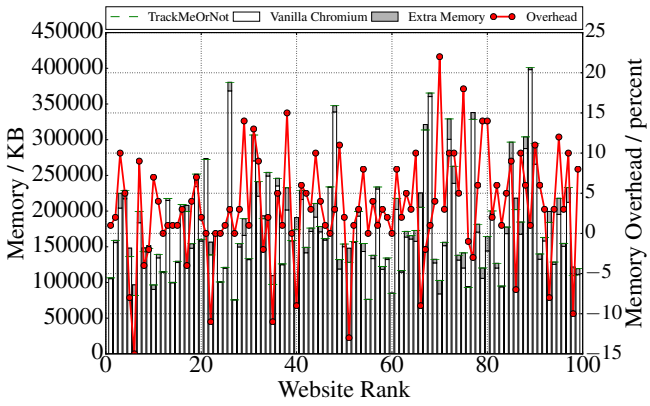


Figure 4.8: Peak memory usage when visiting each of Alexa top 100 US websites under Content Analysis Only Configuration (CAOC).

4.5.2 Memory

`TrackMeOrNot` may require more memory in runtime because it includes 78 classification models (including a large vocabulary of features) in our implementation. In addition, if a browsing context switch is requested, `TrackMeOrNot` needs to create new renderer process which may also increase its memory usage. For these reasons, we measured the peak memory usage of `TrackMeOrNot` and vanilla chromium for 15 seconds in each of the 3 visits to a web page.

Figure 4.8 and Figure 4.9 show the peak memory usage of vanilla Chromium and the extra peak memory usage of `TrackMeOrNot` when visiting the main pages using Content Analysis Only Configuration (CAOC) and Browsing Context Switching Configuration (BCSC), respectively. The memory overhead is the ratio of extra peak memory usage of `TrackMeOrNot` to the peak memory usage of vanilla Chromium. For both configurations, the memory overheads are between -15% and 22%. The means are 3.06% (CAOC) and 1.68% (BCSC), respectively. We observe that sometimes `TrackMeOrNot` consumed less memory than vanilla Chromium, which might be attributed to that smaller dynamic contents were loaded when using `TrackMeOrNot` to visit those web pages. Similarly, the memory over consumed by `TrackMeOrNot` may also due to the dynamics of web contents. As a result, `TrackMeOrNot` did not significantly require more memory than the vanilla build.

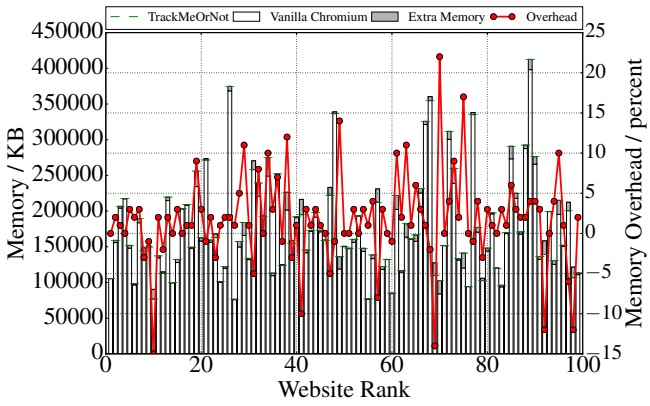


Figure 4.9: Peak memory usage when visiting each of Alexa top 100 US websites under Browsing Context Switching Configuration (BCSC).

4.6 Anti-Tracking Evaluation

As is mentioned in §4.3, `TrackMeOrNot` allows users to specify unwanted page visits based on web content category. Our implementation of `TrackMeOrNot` includes 78 classifiers for categorizing the web pages that users visit. `TrackMeOrNot` compares the output of the classification models with user specified needs and switch a browsing session between different browsing contexts. In this section, we demonstrate how effectively `TrackMeOrNot` uses the classifiers to conceal users' privacy sensitive visits. In particular, we first describe our experimental design. Then, we evaluate how effectively the classifier satisfies user's privacy needs.

4.6.1 Experimental Design

To evaluate the effectiveness of `TrackMeOrNot` on hiding sensitive web browsing activities – such as browsing pornographic web pages and health related forums – we need to simulate a series of web page visits and then examine if `TrackMeOrNot` can accurately conceal user specified privacy sensitive visits when visiting these web pages. As a user may specify any category of visits as privacy sensitive browsing activities, we need a sequence of page visits that covers all spectrum of categories. We selected web pages that had not been used for training any classifier from the AOL data set discussed in §4.4 as our evaluation web page

corpus, which contains 43,807 English web pages in 78 unique categories.

To examine the effectiveness of `TrackMeOrNot` on hiding privacy sensitive visits, we also need to know users' needs to browsing privacy. In other words, we need to know the page categories in which a user is (or not) willing to disclose his or her visits to vendors. To obtain the users' needs to browsing privacy, we conducted an online survey through Amazon Mechanical Turk. The survey was approved by the IRB of our institute. We presented all 78 categories to participants and asked them to choose the page categories in which they are not comfortable to disclose their visits. In addition, participants were asked to choose the page categories in which they are comfortable to share their visits. Ultimately, we collected 145 valid responses. The demographic distribution of the 145 subjects is shown in Table 4.1. Table 4.2 and Table 4.3 show the top categories of web pages that users specified as blacklist rules or whitelist rules, respectively. In other words, they represent the page categories on which users want to hide or disclose their browsing activities from or with online vendors. From our collected questionnaires, we found 14 participants who expressed strong privacy concern and did not want to share any of their visits with vendors. We also observed 2 participants who stated that they were not concerned with privacy and wanted to offer all of their footprints to vendors. Overall, we obtained 129 unique privacy needs from all the participants.

For each unique privacy need, we encode it by converting it into blacklist and whitelist rules as discussed in §4.3.2. We illustrate the number of blacklist and whitelist rules across 129 unique privacy needs in Figure 4.10. Note that, for those page categories that a user does not specify as *"comfortable to reveal"* or *"reluctant to disclose"*, we convert them into either whitelist rules or blacklist rules by assuming the users had specified *persistent* or *anonymous* as their fallback browsing context, respectively. Thus, we have two different rule sets across 129 unique privacy needs (see Figure 4.11 and Figure 4.13).

Including two special whitelist and blacklist rule pairs that indicate *"do not disclose any visits" and "reveal all visits"*, we configure `TrackMeOrNot` using the 256 whitelist and

Table 4.1: Distribution of demographics of survey subjects.

Age					Gender	
18-24	25-34	35-44	45-54	55+	Male	Female
8	58	34	29	16	60	85
5.52%	40.00%	23.45%	20.00%	11.03%	41.38%	58.62%

	High school or less	Associates	Bachelor or higher
Education	32	25	88
	22.07%	17.24%	60.69%

Table 4.2: The top-10 page categories in which users do not want to disclose their visits to vendors.

Category	% of votes
society/sex	78.67
finance	58.67
society/dating	58.67
health and fitness/disorders	52.67
society/crime	51.33
law, govt and politics/armed forces	50.00
law, govt and politics/legal issues	50.00
religion and spirituality	47.33
law, govt and politics/government	47.33
health and fitness/disease	47.33
law, govt and politics/law enforcement	46.00

blacklist rule pairs shown in Figure 4.11 and Figure 4.13. Then, we simulate the visits to the aforementioned 43,807 web pages using `TrackMeOrNot`. We examine the accuracy of hiding blacklist visits and disclosing whitelist visits. In addition, we study the False Negative Rate (FNR) and False Positive Rate (FPR) of our `TrackMeOrNot`. Specifically, a page that needs to be browsed in an anonymous browsing context is a positive sample in our evaluation. The false negative rate is the ratio of number of false negatives (incorrectly predicted as negative) to the number of positives (including false negatives and true positives). The false positive rate is the ratio of number of false positives (incorrectly predicted as positive) to the number of negatives (including false positives and true negatives). The system would mistakenly disclose many privacy sensitive visits to vendors when the false negative rate is high. Similarly, the system may hide many visits that the user feels OK to share with vendors when the false positive rate is high. We present the evaluation results in the next subsection.

Table 4.3: The top-10 page categories in which users are comfortable to share their visits with vendors.

Category	% of votes
arts and entertainment	64.00
food and drink	54.00
hobbies and interests	50.67
sports	45.33
pets	45.33
shopping	44.00
travel	42.67
home and garden	42.67
news	42.00
education	39.33

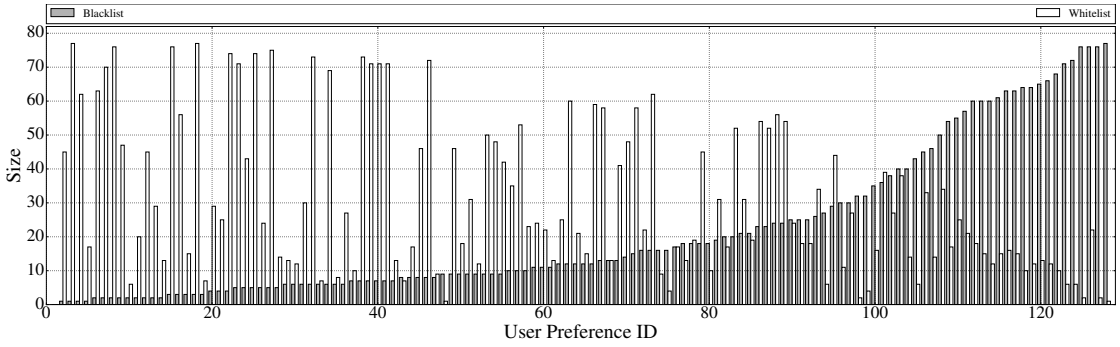


Figure 4.10: The number of whitelist and blacklist rules across 129 distinct privacy needs.

4.6.2 Evaluation Result

Figure 4.12 and Figure 4.14 show the performance of `TrackMeOrNot` in terms of accuracy, false positive rates and false negative rates for persistent and anonymous fallback tracking preferences, respectively. We observed that `TrackMeOrNot` achieved 0.86 accuracy in hiding and disclosing user visits on average for both persistent and anonymous fallback tracking preferences. The minimum accuracies were 0.69 and 0.74 for the two different settings, respectively, where the maximum accuracy in both settings was 1.00. The results indicate `TrackMeOrNot` can effectively cloak user specified privacy sensitive visits and disclose a certain amount of footprints regardless of the fallback browsing context.

We observed some interesting patterns of false positive rates and false negative rates in regards to the number of blacklist rules in the tracking preference. As is evident in

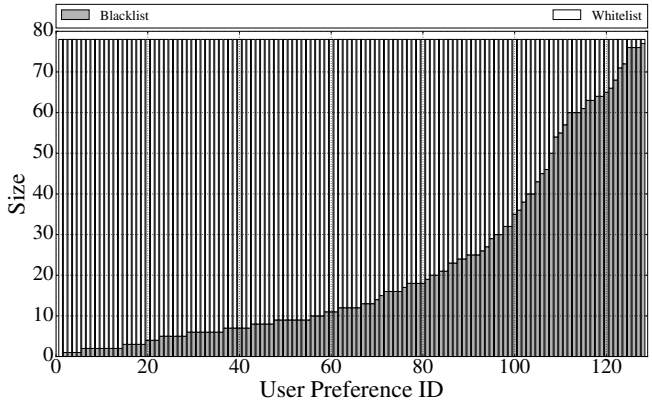


Figure 4.11: Tracking preferences using persistent fallback browsing context.

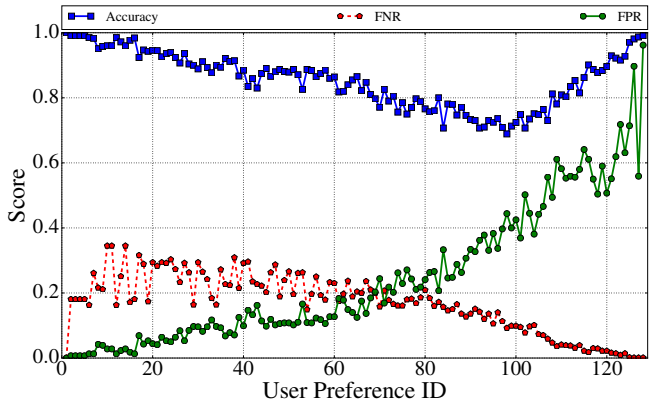


Figure 4.12: Evaluation results on 129 tracking preferences with persistent fallback browsing context.

Figure 4.12 and Figure 4.14, the false positive rates increased when a user specified more page categories as his or her blacklist rules. On the other side, we also observed the decrease in false negative rates as more categories were specified in blacklist. The reason behind these patterns are that with more categories that are specified as blacklist rules, `TrackMeOrNot` relies on more binary classifiers to examine blacklist visits, and consequently a web page is more likely to be classified as positive.

The average false negative rates of `TrackMeOrNot` were about 0.17 and 0.12 and average false positive rates were about 0.24 and 0.36 when using persistent and anonymous fallback browsing context, respectively. As is shown in Figure 4.12 and Figure 4.14, the false negative and false positive rates are complementary. For users concerned more with privacy than personalized user experience, they may configure `TrackMeOrNot` to achieve a low false negative rate and a high false positive rate by specifying more blacklist rules. In contrast,

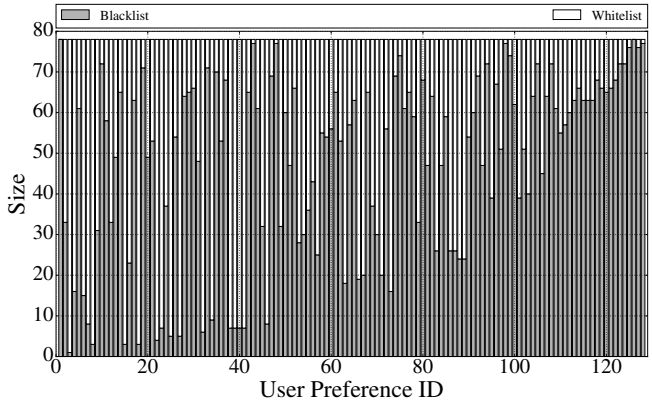


Figure 4.13: Tracking preferences using anonymous fallback browsing context.

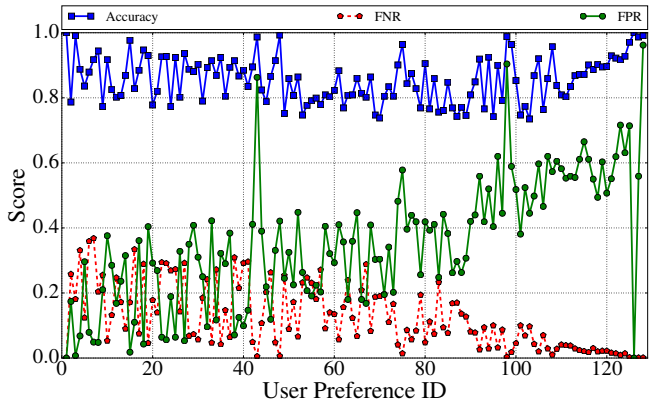


Figure 4.14: Evaluation results on 129 tracking preferences with anonymous fallback browsing context.

users may configure `TrackMeOrNot` with a high false negative rate but a low false positive rate by trading her or his need for privacy for better usability.

4.7 Related Work

Defense against HTTP cookie tracking. HTTP cookie tracking is a stateful tracking technology. It stores in a user's browser a piece of data (including a unique identifier) set by an online vendor while the user is browsing a website that contains the vendor's content. The cookie is automatically sent to the vendor whenever the user visits a website that contains the vendor's content in the future. It can be used to analyze the user's web browsing behavior. Tracking cookies are widely adopted by advertising networks for the purpose of serving up "interest-based" or "behaviorally targeted" ads. To stop them from tracking a person's surfing habits, companies and non-profit organizations (*e.g.*abine [98], NAI [99] and DAA [100])

implement a cookie opt-out mechanism which enables users to block and prevent the advertising network from installing future tracking cookies. A recent study demonstrates many drawbacks of this approach including poor usability and unreliability [26]. As an alternative approach, user self-help anti-tracking tools are developed and implemented [15, 16, 27]. They defend trackers by blocking HTTP requests to corresponding vendors. For example, Adblock plus [15] and Ghostery [16] impede HTTP requests to advertising networks, and thus browsers cannot report user footprints to the advertising networks. Both cookie opt-out and self-help anti-tracking tools are designed for defending HTTP cookie tracking. Considering a number of online vendors have been discovered using advanced technologies to track users [101], the effectiveness of these approaches wanes. In this chapter, our proposed anti-tracking mechanism not only can impede HTTP cookie tracking, but also defend many other previously known stateful tracking technologies, e.g., supercookies.

Defense against Supercookies. Apart from HTTP cookie tracking, another stateful tracking technology is supercookie tracking. Using this technology, online vendors can encode a globally unique identifier – supercookies – into a web browser. For example, vendors can abuse HTML5 local storage feature [102] or Flash Local Share Object [103] to store a unique identifier on a user's hard drive for tracking the user's digital footprints later on. To the best of our knowledge, there are only two approaches that can be adopted to impede such advanced tracking technologies. Private browsing [104] is one of these approaches, which does not store any local data that could be retrieved at a later date. Using private browsing, a user can therefore prevent vendors from using supercookies to track her surfing habits. Another defense against supercookies is TrackingFree [105], an anti-tracking browser that can impede supercookie tracking practice by partitioning a user's visits into multiple isolation units based on URL. With this isolation, online vendors can still store supercookies but not correlate a user's browsing activities across websites. One problem of this approach is that the system overhead linearly increases when a user browses more websites because TrackingFree maintains an isolated browser state for each unique website

and OS needs to allocate a new memory space for each isolated browser state. Considering unexpected memory consumption can potentially jeopardize user experience, our proposed anti-tracking mechanism follows a lightweight design principle which constructs in-memory isolation units based on browser tabs. In addition, our proposed anti-tracking mechanism goes beyond the aforementioned two approaches by allowing users to instruct web browser to selectively block tracking activities. In addition to boosting profits, online vendors use information collected to personalize user experience. From the usability perspective, our anti-tracking mechanism therefore allows users to enjoy customized browsing experience without worrying about privacy invasion, while existing defense restrict data sharing completely and users cannot obtain any benefits from personalization.

Defense against Stateless Fingerprinting. Different from the stateful tracking technologies discussed above, a new class of web tracking technologies can use stateless information to identify users and report their surfing habits. This new class of tracking technologies neither stores nor retrieves data on user's hard drive. Instead, it tracks a user by learning properties of her web browser and forming a unique or nearly unique identifier (*i.e.*, fingerprinting) [86]. To counteract such a tracking mechanism, anti-tracking technologies focus on making browser fingerprints non-deterministic across multiple browsing sessions [106, 87, 107]. This makes vendors difficult to link a user's multiple visits. For example, Nikiforakis *et al.* designed and developed PriVaricator [87] that utilizes the power of browser fingerprint randomization to break vendors' ability to connect the same fingerprint across multiple visits. Our anti-tracking mechanism is orthogonal to defense against stateless tracking. In this chapter, we focus on building an anti-tracking mechanism that impedes stateful tracking based on user demand.

4.8 Summary

In this chapter, we have presented `TrackMeOrNot`, a new tracking control mechanism that allows users to selectively share their online footprints with vendors for better user experience

while shielding privacy sensitive browsing activities from online trackers. TrackMeOrNot provides a user with two browsing contexts – anonymous and persistent context – and a web browser can switch a user's browsing session between the contexts based on user specified privacy needs. We demonstrate how a user can specify his or her privacy need and employ TrackMeOrNot to surf the web without disclosing privacy sensitive visits accordingly.

As TrackMeOrNot allows users to selectively disclose their online footprints, users can enjoy personalized online experience without worrying about privacy. From the perspective of online vendors, TrackMeOrNot may persuade users overly concerned with privacy to share footprints selectively for specific rewards, and vendors may use shared browsing habits to yield more profits.

CHAPTER 5
MONITORING AND UNDERSTANDING WEB CONTENT ACCESS BEHAVIOR OF JAVASCRIPT

5.1 Motivation

JavaScript is broadly used in the creation of modern web applications. Scripts from third-party providers, *e.g.*, social networks, analytic and user tracking services, ad networks, are extensively included by web applications to provide a rich, dynamic and interactive experience to end users. While extending the functionalities of web applications, third-party scripts are particularly disconcerting, especially considering its flexibility and capacity as well as the wealth of sensitive data presented in today's web applications (*e.g.*, credentials, credit card number, and email address). Third-party JavaScript code is granted unrestricted privilege to access an embedding web page that it is included. A malicious third-party JavaScript provider can easily steal authorization cookies and confidential information about a user from the web pages that embed its scripts, or just arbitrarily alter the content in the web pages.

Many security conscious developers would embed scripts only from reputable third-party providers, such as Google and Facebook. By default, however, web browsers allow for additional JavaScript code to be dynamically loaded by any third-party script that is already embedded, if no special restriction, *i.e.*, the Content Security Policy, is configured by the developer. Even if a developer has set up a CSP rule to load remote scripts from a list of trusted third-party providers, the permitted script can still enjoy full access to the content of the web application, which can still put threats to the application and its end users. A web application can be fully controlled by an adversary if a trusted remote provider is compromised. For example, the website of the popular jQuery library was compromised in September 2014 [108], putting millions of websites that embed this library directly from

jquery.com at risk.

A lot of prior work has demonstrated the threats that can be posed by a malicious script [109, 110, 111]. Many systems and tools are also proposed to help limit the functionality of a remote script [112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124]. However, many web developers do not configure security setting for their web applications, because many security mechanisms are only available as prototypes and are difficult for developers to learn. More importantly, many developers think the benefit of embedding a third-party script is greater than the potential harm it can make, because there is no good way to help them *understand* the behaviors of the included third-party scripts. It is very hard to reason about the behaviors of JavaScript code because of its dynamic nature. Both static analysis and dynamic analysis techniques do not work well for JavaScript, because JavaScript code is often obfuscated, dynamically loaded from remote servers at run time, and even dynamically generated on-the-fly. Developers have to trust that the embedded third-party scripts would not abuse their full privilege.

In this chapter, we present `DOM-Logger`, a tool to help web developers monitor and understand the behaviors of third-party scripts that are embedded in their applications. Specifically, `DOM-Logger` logs any JavaScript access to a HTML Element object in the Document Object Model (DOM) tree of a web page. `DOM-Logger` can also track the creation of DOM objects to help reason about what content is inserted by a third-party script in a web page. With `DOM-Logger`, we conduct a comprehensive study on how JavaScript code access web content in general. We seek to answer the following questions in our study:

- Who are the most popular third-party JavaScript providers?
- How much content is generated by third-party scripts in today's web applications in general?
- How many DOM accesses does a script make in a web page?
- How frequently does a script access content created by other scripts?

- Is there any third-party JavaScript code that extensively access sensitive data in web pages embedding it?

Note that we do not focus on analyzing the complete behavior of a specific script in detail in our study. Rather, we aim to empirically investigate the DOM access behaviors of third-party scripts in general. To the best of our knowledge, our study is the first large-scale study that reports the DOM access behaviors of JavaScript in modern web applications.

5.2 DOM-Logger

`DOM-Logger` is a tool for monitoring and logging accesses to HTML DOM Element objects (DOM objects) by JavaScript code. `DOM-Logger` tracks the creation of each DOM object and monitors how the objects are manipulated by JavaScript. We first explain how `DOM-Logger` records DOM access logs (§5.2.1), then discuss how `DOM-Logger` can attribute the creation of an DOM object (§5.2.2) and support monitoring the behaviors of dynamically generated scripts (§5.2.3), and finally describe how we implement `DOM-Logger` in the Chromium browser.

5.2.1 Recording Accesses

An DOM Element object could be accessed through DOM APIs by JavaScript in the following ways. We use the keyword `element` to represent a DOM HTML Element object in the following examples.

1. **Read.** In a `Read` access, the attributes of an element are directly referenced by JavaScript to get the data associated with it. For example, the `textContent` attribute of `element` object is read to the JavaScript variable `text` in the code snippet `var text = element.textContent;`. JavaScript can also call some methods of an object to read data, *e.g.*, `var className = element.getAttribute("className");`.

2. **Write.** In a `Write` access, a JavaScript code directly assigns value to an attribute of an element, *e.g.*, `element.innerHTML = "<div>data</div>";`. Note that besides modifying the value of the `innerHTML` attribute, this code snippet results in that a new element is created and inserted as a child of `element`. Some methods can also be called by JavaScript to modify the attributes of an element, *e.g.*, `element.setAttribute("id", "myID");`. In particular, API calls like `removeChild`, `appendChild` and `replaceChild` on an element object are also recorded as write operation.

3. **Execute.** In an `Execute` access, the methods rather than the attribute members of an element are directly invoked, *e.g.*, `element.scrollIntoView();`. As we show in the above examples, note that by executing some methods the *Read* or *Write* access to the element can also be triggered.

Besides the above three types of direct access, an element object can be indirectly accessed through DOM APIs as well. For instance, the HTML5 `Canvas drawImage()` method takes an `Image`, `Video` or `Canvas` element as argument to draw onto the canvas[1]. The element passed to `drawImage()` is read internally in this method.

`DOM-Logger` is designed to log all direct HTML DOM Element object API calls and indirect DOM object accesses by JavaScript code. In `DOM-Logger`, an access is associated with an access type, which represents the operation on the receiver DOM object, *i.e.*, the object being accessed. Specifically, an access type can be of one or a combination of the three operations: `Read`, `Write`, and `Execute`. In addition to logging access type, we record the member (either an attribute or a method) of an object that is accessed in the log. We also log all accesses to the global `document` object, because sensitive APIs such as `Cookies`, `createElement` are invoked through the `document` object.

To attribute a DOM access to a script, we need to obtain the identity of the accessing JavaScript code. In HTML, JavaScript code is usually inserted between <script> and </script> tags as inline script, or stored in an external JavaScript file and loaded with

[1] http://www.w3schools.com/tags/canvas_drawimage.asp

<script> tags as external script There are other types of inline JavaScript code. For example, JavaScript code can be written as the event handler attributes of HTML elements. In `DOM-Logger`, each <script> element is assigned with an unique `scriptID`. An inline script that is not wrapped within a <script> tag is also assigned with an unique scriptID. Further, a script is identified by its `sourceURL`, which is the URL the browser uses to load the remote JavaScript. Note that the sourceURL of inline scripts is an empty string. We use the URL of the embedding frame, *i.e.*, the URL that the browser uses to load the HTML document in the embedding frame, as the sourceURL of inline scripts. We will discuss how we attribute a DOM access to an inline script that is dynamically generated in §5.2.3. Besides the scriptID of the accessing script, we also record the row number, column number and the function name of the specific accessing JavaScript code in one script.

5.2.2 Tracking Element Creation

HTML elements are normally statically written in the HTML source file, parsed as DOM Element objects and inserted into the DOM tree by the browser parser. As the `innerHTML` example we showed in §5.2.1, HTML elements can also be created on-the-fly by JavaScript through APIs such as `document.write("<div>...</div>")`, and `document.createElement("div")`.

One key functionality of `DOM-Logger` is attributing the creation of one HTML element object to a script. This is very useful for understanding how a script access different objects that are created by itself, by the document owner (if the script is not the owner's script), and by other scripts. Specifically, we attach a hidden `initiator` attribute to each DOM object in `DOM-Logger` to represent the creator of the object. The initiator attribute is the scriptid of the script that has created the corresponding DOM object. All parser-created static objects are assigned a special initiator value – `0` – to represent the owner of the document, *i.e.*, the *first-party*. The initiator of dynamically created objects is the scriptID of the JavaScript code that calls the corresponding element creation API.

5.2.3 Monitoring Dynamically Generated Scripts

Similar to HTML elements, JavaScript code can also be dynamically generated in web applications. Specifically, as HTML elements, new <script> elements can be dynamically created by JavaScript using the same APIs for creating elements. String can also be parsed as JavaScript code if its an inline event handler or through the API `window.eval("...")`.

Getting the sourceURL of a dynamically inserted <script> element that loads an external script from a remote host is not different from getting the sourceURL of one static <script> element. However, it is not straightforward to get the sourceURL of other dynamically generated scripts because their sourceURL is blank. To overcome such difficulty, we hook the APIs that are used to generate dynamic scripts. The sourceURL of the JavaScript code that is calling the script generation API is used as the sourceURL of the newly generated inline script. For an inline event handler, we search in the access logs of the receiver object to find the last script that sets the event handler as the generating script (parent script). If no such an entry is found, the parent script is the one that creates the receiver object and sets the inline event handler.

To distinguish the dynamically generated script or the *child* script (either an inline script or an external script), from the generating script or the *parent* script (the one that generates the script), we also record the scriptID of the parent script as the `parentScriptID` attribute of the child script. The parentScriptID of all static scripts that are initially embedded by the document owner is set to `0`.

5.2.4 Implementation

`DOM-Logger` is implemented by modifying the Chromium browser. We decide to implement `DOM-Logger` in a full-fledged web browser because some websites may not render correctly in a headless browser or a simpler user client. Some scripts may even attempt to hide some operations from uncommon user agents. We inserted access monitoring code in the C++ implementation of the V8 binding layer between the V8 JavaScript engine and the DOM

implementation in WebKit. When a DOM element is accessed by JavaScript, the access monitoring code is executed with the C++ method that implements the corresponding DOM API. The access monitoring code identifies the JavaScript caller by fetching its scriptID, and then appends the access log to the hidden `accessLog` attribute of the DOM object. We inserted the access monitoring code into all the methods that implement a DOM API of all sub classes of the `Node` class in WebKit[2].

To track the dynamic creation of new elements, we insert additional logging code into methods that are used to create new DOM objects. This code is used to set the initiator attribute of the element that is dynamically created. Similarly, we also insert custom logging code into methods that compile and execute JavaScript in WebKit to monitor scripts that are dynamically generated. This code sets the parentScriptID attribute of dynamically generated script and also logs that information in the global Document object. Furthermore, the sourceURL of all scripts is stored in the Document object when a script object is first created in WebKit.

The `accessLog`, `initiator`, `scriptID` and `parentScriptID` attributes of DOM nodes, and the `scriptID-to-sourceURL` and `scriptID-to-parentScriptID` dictionaries (also implemented as hidden attribute members of the Document object) can be read by JavaScript for further analysis. For performance concerns, we implement a lazy update mechanism for setting the above attributes. The values of these attributes are kept in the hidden attribute members of the modified C++ classes. They are not updated in the DOM tree until the attributes are first accessed by JavaScript.

5.3 Collecting DOM Access Logs in the Real World

To understand the behaviors of JavaScript in the real world, we performed a large-scale measurement experiment of the Alexa top 100,000 websites with `DOM-Logger`. We first describe the experiment setup (§5.3.1), then discuss how we group logs of a script collected

[2]Document and all DOM Elements are sub classes of Node.

from multiple websites (§5.3.2), and highlight some statistics of the crawled data (§5.3.4).

5.3.1 Setup

We aim to gather a large collection of DOM access logs from web applications in the real world. Such data set can help understand how JavaScript code accesses web content in modern web applications. In particular, we are interested in looking for suspicious or even malicious accesses that are made by third-party JavaScript code. We used our DOM access logging tool `DOM-Logger` to perform a large scale web crawl in May 2017. Since `DOM-Logger` is implemented in a full-fledged web browser, we are not able to run hundreds of `DOM-Logger` crawler in parallel.

For each website in our data set, we used a browser automation tool – Selenium[3] – to drive `DOM-Logger` to browse its main page for up to three times. In each navigation attempt, `DOM-Logger` waits for up to 30 seconds till the web page is fully loaded or the timer expires, whichever comes first. After that, we then executes a script in `DOM-Logger` to traverse the DOM tree to instruct `DOM-Logger` to update all the access logs and meta data it records in the DOM tree. Specifically, this script reads the `accessLog`, `initiator`, `scriptID` and `parentScriptID` attributes of each DOM node[4] it visits, and the `scriptID-to-sourceURL` and `scriptID-to-parentScriptID` dictionaries of the <document> node. Finally, we use Selenium to save the updated DOM tree as a HTML source file on disk for further analysis. The two dictionaries and the `accessLog` of the <document> node are stored in a separate file, because the Document class is not a HTMLElement.

Our crawl method has several limitations. First, the DOM access logs of only the main page of a website are collected in our web crawl. Many websites may load a different set of scripts in other web pages. Nevertheless, our goal is to obtain a preliminary data set that

[3] http://www.seleniumhq.org/

[4] The `scriptID` and `parentScriptID` attributes are only accessed if a DOM node is an instance of <script> element.

helps understand the behaviors of JavaScript in general, rather than to exhaustively cover all scripts that may be embedded on all web pages on the Internet. We believe that our crawl of 100,000 web pages is sufficient for achieving our goal. We plan to perform additional crawls of more web pages of a website in our future work. Second, we do not interact with each visited web page as a real user would do. As a result, many JavaScript functions that listen for specific user interactions may not be executed in our visit. Furthermore, some scripts might have not been loaded because they also depend on user generated events. Although we cannot cover all execution paths, our measurement can collect the *default* behavior of a script, *i.e.*, the access logs that are common among all visitors. Indeed, achieving full code coverage is extremely difficult (if not impossible) because of the dynamic nature of JavaScript [125, 126, 127, 128, 129].

5.3.2 Grouping Logs

A script object is uniquely identified by its scriptID in a web page. This unique identifier of a specific script is not consistent across different web pages and even across different visits to one web page. To group the logs associated with one script collected on different web pages, we use the sourceURL of the script as its global unique identifier. However, multiple scripts with distinct scriptIDs may be mapped to the same sourceURL even on a single web page. For example, we assign the sourceURL of a parent script to the sourceURL of a dynamically generated inline script as in §5.2.3 We think such mapping is acceptable because this set of scripts is written or embedded by the same party. Actually, the inline scripts can be merged within the external script to represent a group of JavaScript code. Besides the sourceURL (URL in short) of a script object, we also group DOM logs by the `Origin` and the `Domain Name` (Domain in short) of a script. Such grouping methods allow us to characterize scripts in different granularities.

5.3.3 Privileges of Scripts

Although all scripts loaded in one frame have the same privilege as the first-party scripts in today's web applications, we think that by default any third-party script should be not trusted and be "granted" with lower privilege. However, we find that many web applications embed their own scripts from separate origins other than the first-party origin. For example, Google embed one script from the origin https://apis.googlecom on its main page https://www.google.com/. Hosting static content on *separate hosts* to improve page load time is a common practice in the development of modern web applications. For web pages like Google's main page, we might use the domain of a script to test whether it is owned by the first-party and should be trusted. We do not think, however, such owner-operated third-party scripts shall be given the full privilege as first-party scripts, because they might be embedded across multiple origins and their hosting servers might be compromised one day. What is worse, some websites host their scripts on hosts of *separate domains*, which are the domains of the Content Delivery Networks (CDNs) that are operated by the websites. For instance, Amazon include many scripts from the origin https://images-na.ssl-images-amazon.com on its main page https://www.amazon.com/; several scripts on the main page of Bank of America https://www.bankofamerica.com/ are loaded from the origin https://www2.bac-assets.com. Similarly, we think such scripts loaded from other domains shall be granted with limited privilege.

To this end, we classify scripts loaded in one page into four privilege levels as follows. Note that a smaller numeric value indicates a higher privilege.

Privilege-0 (Priv-0): Scripts in this privilege level are the first-party scripts and are fully trusted.

Privilege-1 (Priv-1): Scripts in this privilege level are the owner-operated third-party scripts that are loaded from the same domain of the current web page.

Privilege-2 (Priv-2): Scripts in this privilege level are the owner-operated third-party

scripts that are loaded from a different domain of the current web page.

Privilege-3 (Priv-3): Scripts in this privilege level are the third-party scripts that are not operated by the owner of current web page and shall not be trusted.

Some websites do not have their own CDN infrastructure and ask third-party CDN providers to manage their JavaScript code. One may argue that such scripts shall be classified to Priv-2 instead of Priv-3. We conservatively classify such scripts to Priv-3 because the uploaded scripts might be modified by a compromised or malicious third-party CDN provider.

One important measurement goal of this chapter is to compare the behaviors of the third-party scripts from those of the first-party scripts. We need to correctly assign a privilege level to one third-party script because it may be operated by the website owner. We can easily detect Priv-0 and Priv-1 scripts from their URLs. As we discuss above, however, we cannot simply use the URL, origin or even domain name of a third-party script to decide whether a third-party script shall be classified to Priv-2 or Priv-3.

To solve this problem, we propose a new technique to determine whether two scripts with distinct domains are operated by the same organization. Specifically, we inspect the email addresses in the DNS SOA records[5] of the two scripts' host names. We leverage the DNS SOA records for this task because many organizations would use the same email address to register multiple domains. For instance, the email address fields in the DNS SOA records of domain names fbcdn.com and facebook.com are both *dns@facebook.com*, respectively. However, there is one caveat in using the above simple technique to determine the organization of a URL. Different organizations may use the same Managed DNS providers[6]. As a result, the email addresses in their SOA records are the same. For example, the SOA email addresses of both instagram.com and netflix.com are *awsdns-hostmaster@amazon.com*. To overcome the limitations of using email address alone as the identifier of one organization, we supplement the SOA record with the name servers in the

[5]https://en.wikipedia.org/wiki/SOA_Resource_Record
[6]https://en.wikipedia.org/wiki/List_of_managed_DNS_providers

DNS NS record[7] of a host or domain name. In order to be classified to Priv-2, a third-party script shall first have the same SOA email address of the web page that it is embedded in. Secondly, the third-party scripts and the first-party web page shall have common *domain names* of their name servers. We do not strictly require they have common name servers because the name servers can be dynamically selected from a large pool. Furthermore, we compile a list of known managed DNS providers and dynamic DNS providers as a blacklist to filter name servers of those providers. Only if the common name server(s) is not owned by a public DNS provider shall the third-party script be classified to Priv-2.

Using our technique, we successfully identified Priv-2 scripts on many websites, *e.g.*, scripts loaded from https://images-na.ssl-images-amazon.com on https://www.amazon.com/. Our technique may incorrectly classify one owner-operated third-party script to Priv-3 in two cases. First, an organization may use one separate email addresses for one of its domain. For instance, although Twitch is now a subsidiary of Amazon, we classified scripts loaded from https://www.amazon.com on https://www.twitch.tv/ to Priv4, because the SOA email addresses of amazon.com and twitch.tv are *root@amazon.com* and *admin@justin.tv*, respectively. Second, an organization may use a managed DNS provider for two or more of its domains. We were not able to automatically find Priv-2 scripts on one of such domains. We consider it as a limitation of our current approach.

5.3.4 Crawling Results

We summarize some of the crawling results in this subsection, and will present more analysis in detail in §5.4.

Overview

Our crawl of the Alexa top 100,000 websites using `DOM-Logger` is completed in about one week on a 32-core CPU server with 256 GB memory. In total, we were able to collect

[7] https://en.wikipedia.org/wiki/List_of_DNS_record_types#NS

data from 96,233 websites (96.3%). We failed to load or save any data from the other 3,777 websites. We manually tried to navigate to some of these websites using a non-modified Chromium browser but could not load any content in 30 seconds, either. We did not observe any DOM access log for 4,833 out of the 96,233 (5.0%) websites. Through manual inspection, we find that these websites either included no JavaScript code (*e.g.*, http://www.parsiteb.com/ and http://www.websiteseguro.com/) or/and served blank content (*e.g.*, http://www.flowsoft7.com/ and http://www.onecount.net[8]) when we used `DOM-Logger` to browse them.

Statistics about Scripts

In our crawl, we have found 422,713 unique scripts (by URL) that are loaded from 134,487 origins of 105,092 domains. We tried to generate a CDF figure of the script distribution by number of websites that one script is included. But we could only observe the long tail in the figure. The top-10 JavaScript URLs, origins and domains are shown in Table 5.1, Table 5.2 and Table 5.3, respectively. The top-10 scripts and origins all belong to Google and Facebook. In particular, Google takes 9 of 10 seats in both the top-10 scripts and top-10 origins. The top-8 origins are still owned exclusively by Google and Facebook. On average, a JavaScript URL (script), origin and domain is found on 2.22, 4.45 and 5.07 websites, respectively. The 90th percentiles of number of websites that a script, origin and domain is included are 2, 3 and 2, respectively. In other words, 90% of scripts, origins and domains are found on at most 2, 3 and 2, respectively. The 90th percentiles are smaller than the averages, suggesting that the third-party scripts on the Internet are dominated by very few top companies, such as Google and Facebook.

We present the CDF figures of the website distributions in terms of number of scripts, origins and domains in each privilege level on one website in Figure 5.1, Figure 5.2 and Figure 5.3, respectively. Note that the domain of Priv-1 scripts is identical to the domain

[8]Only the string *running* is enclosed within <body> in the saved HTML source file.

Table 5.1: The top-10 JavaScript URLs.

Rank	Script	Number (%) of websites
1	https://www.google-analytics.com/analytics.js	50607 (52.59%)
2	https://connect.facebook.net/en_US/fbevents.js	14211 (14.77%)
3	https://pagead2.googlesyndication.com/pagead/osd.js	13764 (14.30%)
4	https://securepubads.g.doubleclick.net/gpt/pubads_impl_117.js	11014 (11.45%)
5	https://www.google-analytics.com/ga.js	10982 (11.41%)
6	https://www.googletagmanager.com/gtm.js	9698 (10.08%)
7	http://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js	8307 (8.63%)
8	http://pagead2.googlesyndication.com/pagead/osd.js	6390 (6.64%)
9	http://www.googletagservices.com/tag/js/gpt.js	6302 (6.55%)
10	http://www.googletagmanager.com/gtm.js	6109 (6.35%)

Table 5.2: The top-10 JavaScript origins.

Rank	Origin	Number (%) of websites
1	https://www.google-analytics.com	59888 (62.23%)
2	https://connect.facebook.net	24696 (25.66%)
3	http://pagead2.googlesyndication.com	15759 (16.38%)
4	https://pagead2.googlesyndication.com	14501 (15.07%)
5	https://securepubads.g.doubleclick.net	11088 (11.52%)
6	https://www.googletagmanager.com	9698 (10.08%)
7	https://ajax.googleapis.com	9542 (9.92%)
8	https://apis.google.com	9295 (9.66%)
9	http://ajax.googleapis.com	8285 (8.61%)
10	http://www.googletagservices.com	6312 (6.56%)

of the first-party, such that there is no Priv-1 domain in our result. We observe on average more Priv-3 scripts (URLs) than Priv-0 scripts, and rarely Priv-1 and Priv-2 scripts – the average number of scripts (URLs) from Priv-0 to Priv-3 are 3.13, 0.30, 0.01 and 6.80, respectively. However, few websites include lots of scripts so the mean might not be a very good metric. The median number of scripts in the four privilege levels are 3, 0, 0 and 5, respectively. The 99th percentile of number of scripts (URLs) from Priv-0 to Priv-3 are 11, 5, 0 and 29, respectively. In other words, 99% of the websites have at most 11, 5, 0 and 29 scripts in Priv-0, Priv-1, Priv-2 and Priv-3, respectively. In particular, http://www.aoji.cn/ and http://www.56products.com/ had more than 60 first-party scripts (Priv-0); http://www.iqilu.com/ and http://www.qingdaonews.com/ embedded 65 and 45 Priv-1 scripts, respectively; https://www.terra.com.br/, http://www.ctrip.com/ and https://www.terra.com/ included 21 Priv-2 scripts; more than 60 Priv-3 scripts were found on http://www.celebzz.com/, http://peopleenespanol.com/ and http://lokalavisen.dk/.

Besides the number of scripts in each privilege level on one web page, we are also interested in the percentage of scripts in each privilege level. We show the CDF figures of

Table 5.3: The top-10 JavaScript domains.

Rank	Domain	Number (%) of websites
1	google-analytics.com	63180 (65.65%)
2	facebook.net	28640 (29.76%)
3	googlesyndication.com	21017 (21.84%)
4	ajax.googleapis.com	17322 (18.00%)
5	googletagmanager.com	15744 (16.36%)
6	google.com	15636 (16.25%)
7	doubleclick.net	13383 (13.91%)
8	googletagservices.com	10852 (11.28%)
9	twitter.com	9279 (9.64%)
10	yandex.ru	5964 (6.20%)

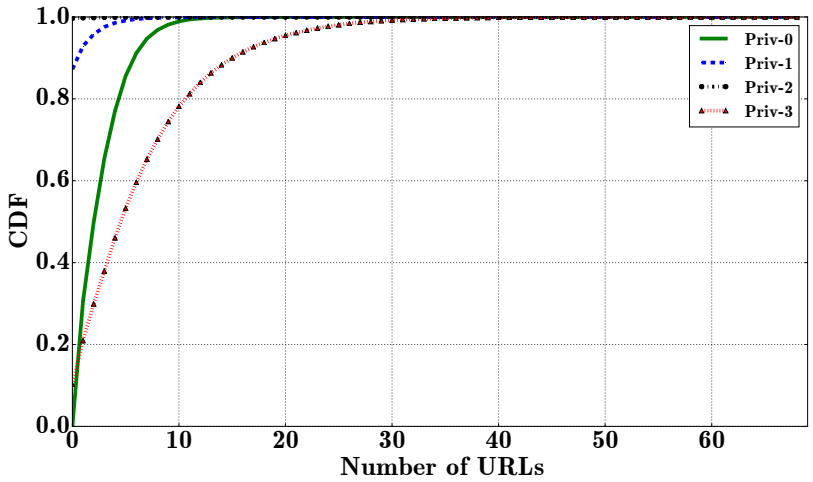


Figure 5.1: Website Distribution by Number of Scripts in Each Privilege.

the website distributions in terms of percentages of scripts, origins and domains in each privilege level on one website in Figure 5.4, Figure 5.5 and Figure 5.6, respectively. On average, 41%, 3%, 0% and 56% of scripts (JavaScript URLs) on a website are in Priv-0, Priv-1, Priv-2 and Priv-3, respectively. Note that the way we calculate the average percentage of a script is different from calculating the average number. In particular, we first calculate the percentage of scripts in each privilege level for each website, then calculate the average percentage values of each privilege across all the websites. The median percentage of scripts in the four privilege levels are 0.33, 0, 0 and 0.61, respectively. We find that the main pages of top websites such as https://www.facebook.com/ and https://www.wikipedia.org only included first-party scripts (Priv-0). On the other hand, more than 98% of scripts on main pages of websites such as http://peopleenespanol.com/, http://fortune.com/, http://www.goodtoknow.co.uk/ and http://time.com/ were Priv-3 scripts. Many of those Priv-3

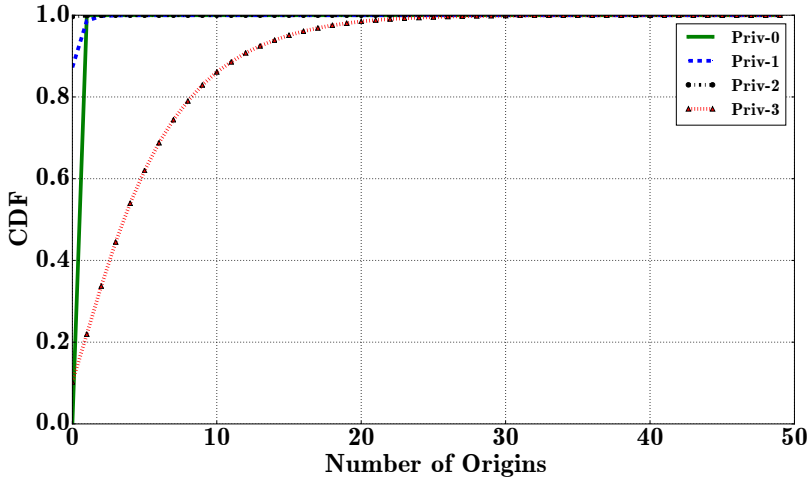


Figure 5.2: Website Distribution by Number of Origins in Each Privilege.

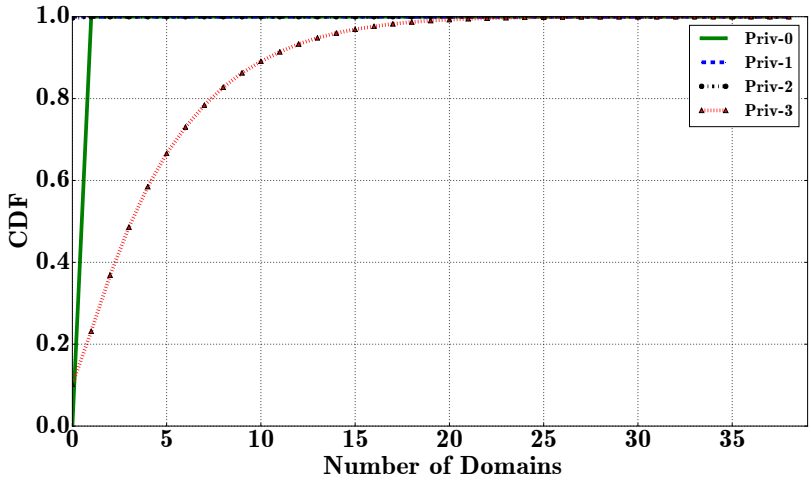


Figure 5.3: Website Distribution by Number of Domains in Each Privilege.

scripts were dynamically inserted by other Priv-3 scripts.

5.4 Characterizing DOM Access behaviors of JavaScript

In this section, we analyze how JavaScript code accesses DOM nodes. In particular, we show that third-party scripts in Priv-3 exhibit very different behaviors from the first-party operated scripts (Priv-0, 1, 2). We even find that many third-party scripts, including those provided by the top websites such as Google and Facebook, were "abusing" their full privilege. First, we present the attribution of DOM element creations to JavaScript (§5.4.1). Then, we study how JavaScript accessed DOM nodes in general (§5.4.2). Finally, we identify scripts that

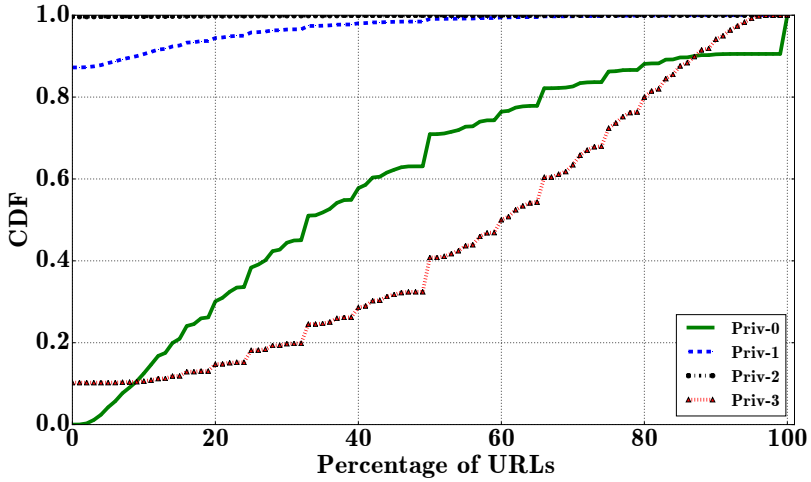


Figure 5.4: Website Distribution by Percentage of Scripts in Each Privilege.

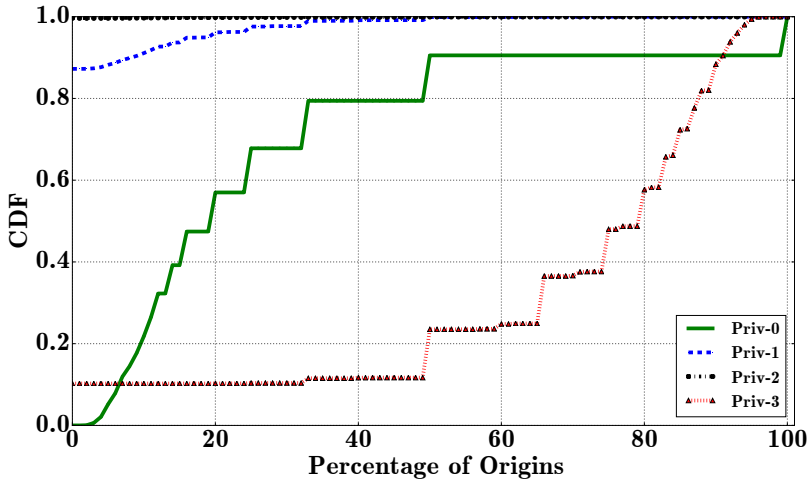


Figure 5.5: Website Distribution by Percentage of Origins in Each Privilege.

accessed DOM nodes created by other scripts (§5.4.3), and scripts that accessed sensitive content (§5.4.4).

5.4.1 Initiators of DOM Elements

We are interested in how much content in a web page is generated by the first-party and by third-party scripts. In particular, we would like to find third-party scripts that create lots of elements on many websites. We use the `initiator` attribute generated by `DOM-Logger` in this analysis. Note that in this analysis we also include the 4,833 websites on which we do not find any DOM access log. All DOM nodes on these websites were statically created by

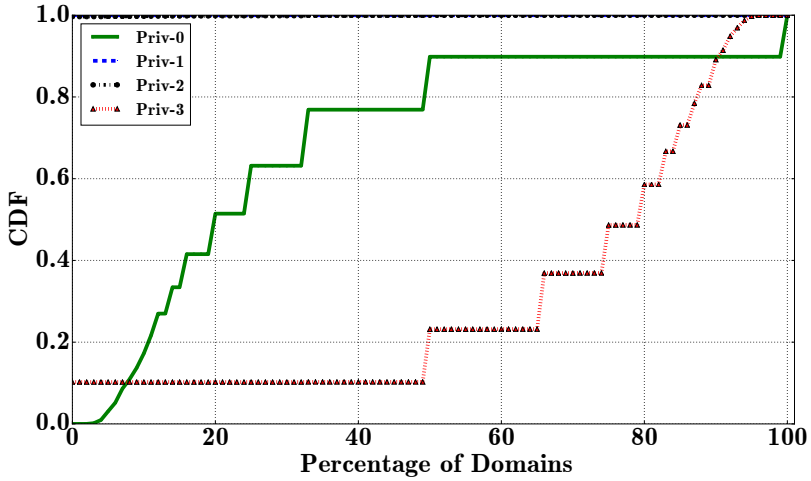


Figure 5.6: Website Distribution by Percentage of Domains in Each Privilege.

the first-party (Priv-0), *i.e.*, the HTML elements were only generated by the browser DOM parser when a web page was first rendered.

On average, we found 973 DOM nodes across the main pages of the 96,233 websites. The 25th, 50th, 75th, 90th and 99th percentiles of number of elements are 315, 692, 1234, 2053 and 5095, respectively. http://www.hubstatic.com/ is the top website that had 40,051 elements. We present the number of DOM elements created by scripts in each privilege level in Figure 5.7. Note that we perform the transformation – $-log10(1-Y)$ – to the Y-axis to highlight the Y values that are close to 1. On average, 899.69, 9.85, 0.36 and 63.12 DOM elements were created by Priv-0, Priv-1, Priv-2 and Priv-3 scripts, respectively. In particular, http://www.hubstatic.com/ is also the top website by number of Priv-0 elements (40,051); https://www.easymarkets.com/int is the top website by number of Priv-1 elements (12,336); 4,414 Priv-2 elements make https://www.walgreens.com/ the number website by number of Priv-2 elements. Surprisingly, 26,280 DOM elements (97.8%) were initiated by Priv-3 scripts on http://www.sofascore.com/, making it the top one by number of Priv-3 elements. The percentage of DOM elements created by scripts in each privilege level is shown in Figure 5.7. The means of percentage of DOM elements in the four privilege levels are 93.60%, 0.67%, 0.03% and 5.70%, respectively. At least 93% of DOM elements on 75% of websites were Priv-0 elements, showing that third-party JavaScript do not add much new

content to a website in general.

We also calculate the average number of elements one script had created. Only 212,501 out of the 422,713 scripts had created at least one DOM element in our crawl. On average, a script initiated 416.76 elements on one web page. However, the median is merely 19. Only 25% of scripts had initiated more than 560 elements on a web page on average. After filtering scripts that were embedded in less than 100 websites, we find https://static.addtoany.com/menu/page.js as the top script by average number of created elements (610.68), followed by https://an.yandex.ru/resource/context_static_r_1853.js (324.20) and https://ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.min.js (273.79). Note that the number of elements initiated by third-party libraries such as jQuery depends on the caller that calls functions in the libraries. One may suggest that we shall attribute elements that are created by such a library to the caller rather than the library. We think it is an open problem. The reason we set the library as the initiator is that the direct caller of the library may be called by a function of another script. However, the bottom function in the JavaScript call stack that initiates the function calls might not expect some intermediate function to call jQuery to create a new element. Furthermore, after a function of a library is called, the library itself can also call its other functions to perform arbitrary operations, including creating new elements. Thus, we set the script of the top (last) function frame in the call stack as the initiator of the newly created element.

5.4.2 DOM Accesses

We have found that third-party scripts do not create much new content in web pages embedding them. In this section, we are interested in understanding how JavaScript access DOM nodes in general. Specifically, we focus on the access type, *i.e.*, Read (R), Write (W) or Execute (X), of each access. As we discussed in §5.2.1, an Execute access may cause a DOM object being read or written as the same time. We treat such an access as either a Read or a Write access in stead of an Execute access.

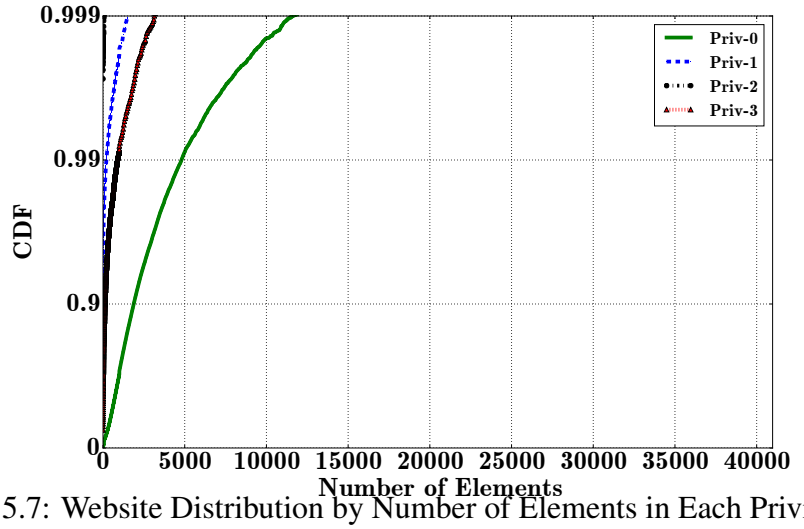


Figure 5.7: Website Distribution by Number of Elements in Each Privilege Level.

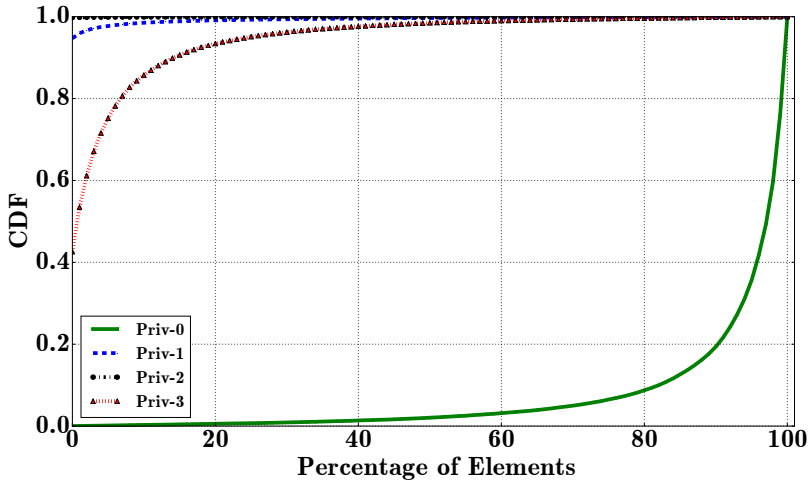


Figure 5.8: Website Distribution by Percentage of Elements in Each Privilege Level.

The average and median number of the three types of accesses made by scripts in the four privilege levels across the 91,400 websites are summarized in Table 5.4. Putting all scripts together, on average they made 17,710 Read, 687 Write and 1,009 Execute accesses to DOM nodes on one web page. Since there are much fewer Priv-1 and Priv-2 scripts, the average and median numbers of accesses by those scripts are significantly lower than those by Priv-0 and Priv-3 scripts. We focus on comparing Priv-3 scripts to Priv-0 scripts. On average, neither Priv-0 nor Priv-3 scripts performed much Write and Execute operations on DOM nodes, compared to Read operations. However, we are surprised to find that third-party scripts in Priv-3 performed nearly 80% more Read accesses than first-party

Table 5.4: Numbers of Accesses by Scripts in Four Privilege Levels across 91,400 Websites

Privilege	Read		Write		Execute	
	Mean	Median	Mean	Median	Mean	Median
0	6,049.75	636	342.45	47	474.74	14
1	791.42	0	45.30	0	67.63	0
2	26.98	0	1.66	0	2.24	0
3	10,842.31	1025	297.31	72	464.65	21

scripts on average. Adding Write and Execute accesses, Priv-3 scripts made up 59.80% of DOM accesses across all the websites. Note that in §5.4.1 on average there were only 5.70% DOM elements that were created by Priv-3 scripts. This result suggests that Priv-3 scripts accessed mainly first-party content rather than their own content. Such high privilege can be easily abused to read and even modify sensitive first-party content. We will present the analysis on cross-owner and over-privileged accesses in §5.4.3.

The median numbers in Table 5.4 are an order of magnitude smaller than the means, indicating that significant number of accesses was observed on very few web pages. For each web page, we also calculated the percentage of DOM accesses made by scripts in each privilege level. The CDF figure is depicted as in Figure 5.9. On average on a web page, 45.30%, 4.90%, 0.14% and 49.66% of the accesses were performed by Priv-0, Priv-1, Priv-2 and Priv-3 scripts, respectively. The median percentages of both Priv-1 and Priv-2 scripts are 0. The medians of Priv-0 and Priv-3 scripts are 33.87% and 51.15%, respectively. This result also demonstrates that the majority accesses were made by Priv-3 scripts.

Table 5.5 lists the top-10 websites that were mostly accessed by Priv-3 scripts. The third column shows the number of DOM nodes on this website. We present the number and percentage of accesses made by Priv-3 scripts on each of the website in the fourth column. Note that we do not include accesses by scripts in a higher privilege level in the calculation. On these 10 websites, over 99% accesses were made by Priv-3 scripts. We also check which Priv-3 scripts performed such significant number of accesses on each website. We find that the script http://w.sharethis.com/button/async-buttons.js appears as the top accessing Priv-3 script on 6 websites. It made up more than 98% Priv-3 accesses on 5 websites and 94% accesses on one website. Similarly, more than 93% Priv-3 accesses were made by one single

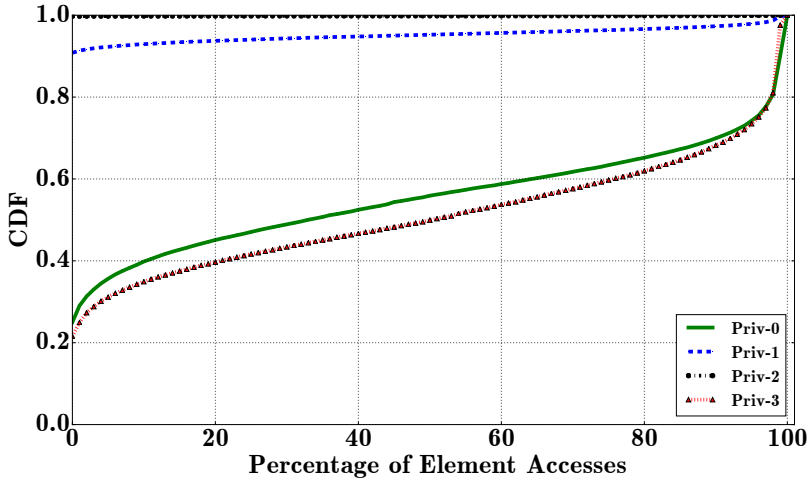


Figure 5.9: Website Distribution by Percentage of Element Accesses by Scripts in Each Privilege Level.

Table 5.5: Top-10 Websites Accessed By Priv-3 Scripts.

Rank	Website	Number of Elements	Number of Priv-3 Accesses
1	http://www.houstonmethodist.org	1185	880040 (100%)
2	http://www.catholicculture.org	947	869789 (100%)
3	http://www.jtsstrength.com	812	851977 (100%)
4	https://www.cashstar.com	979	841391 (100%)
5	http://www.wirebat.com	1087	832846 (100%)
6	http://www.mp4d.com	3399	823251 (100%)
7	http://horrorfreaknews.com	2242	820903 (99%)
8	http://www.siggraph.org	793	811867 (100%)
9	http://www.nfpa.org	978	789928 (100%)
10	http://www.fnews.gr	1159	783494 (99%)

script on the other 4 websites.

To discover the top accessing scripts, we also count the average number of DOM accesses by each script and show the top-10 scripts in Table 5.6. Scripts that are embedded in less than 100 websites are not shown. The third column is the number of websites that one script was included. The *async-buttons.js* script for implementing the social widget of ShareThis loaded from the host *w.sharethis.com* is the top Priv-3 script in terms of average number of DOM accesses. We cannot find a good excuse for needing to make more than 136K accesses on average for it.

5.4.3 Cross-Owner and Over-Privileged Accesses

We study how a script access content created by other scripts in this section. Specifically, we define two types of such accesses by one script: `cross-owner` access and

Table 5.6: Top-10 Priv-3 Scripts by Average Number of DOM Accesses.

Rank	Script	Num. Websites	Average Num. Accesses
1	https://ws.sharethis.com/button/async-buttons.js	108	160066.56
2	http://w.sharethis.com/button/async-buttons.js	466	136770.57
3	http://cdn.viglink.com/api/vglnk.js	149	86684.05
4	http://load.sumome.com/	339	71050.01
5	https://load.sumome.com/	241	42914.30
6	http://s1.adform.net/banners/scripts/adx.js	204	34739.58
7	http://cdn.revcontent.com/build/js/rev2.min.js	136	26846.01
8	http://widgets.outbrain.com/outbrain.js	310	22454.04
9	https://widgets.outbrain.com/outbrain.js	105	21754.16
10	http://cdn.doubleverify.com/avs674.js	161	16297.48

Table 5.7: Numbers of Cross-Owner Accesses across 86809 Websites.

Privilege	Read		Write		Execute	
	Mean	Median	Mean	Median	Mean	Median
0	6245.23	437	334.22	24	485.49	16
1	744.27	0	30.74	0	61.70	0
2	25.23	0	1.26	0	1.99	0
3	10898.29	1164	134.47	24	430.27	19

`over-privileged` access. Cross-owner access is an access to a DOM element initiated by JavaScript with different script (URL), origin or domain. Over-privileged access is an access to a DOM element in a `higher` privilege. We allow scripts in a higher privilege to access content in a lower privilege. Note that we do not study access to Cookies in this section.

The mean and median numbers of cross-owner DOM accesses by scripts in the four privilege levels across 86,809 websites are listed in Table 5.7. It might first look strange that Priv-0 scripts also made cross-owner accesses. Remember that the way we determine an access as cross-owner access is by comparing either the URL, origin or domain of the script and that of the initiator of the accessed DOM object. An access by the Priv-0 script https://www.a.com/lib.js to one first-party object whose initiator is http://www.a.com/ is determined as a cross-owner access. This access, however, is not an over-privileged DOM access. As a result, we observe on average over 6,245 cross-owner Read accesses by Priv-0 scripts on one website. Priv-3 scripts even made over 10,898 cross-owner Read accesses on one website on average. Priv-1 and Priv-2 scripts did not perform much cross-owner accesses, because only few scripts on the 86K websites were in the two privilege levels. An interesting observation is that Priv-0 scripts did more cross-owner Write accesses than Priv-3 scripts. One possible explanation is that many DOM elements were dynamically inserted by

Table 5.8: Numbers of Over-Privileged Accesses to Priv-0 Elements across 82327 Websites.

Privilege	Read		Write		Execute	
	Mean	Median	Mean	Median	Mean	Median
1	762.35	0	31.20	0	63.76	0
2	25.80	0	1.29	0	2.07	0
3	10026.78	1343	128.79	25	374.45	12

Table 5.9: Numbers of Cross-Owner and Over-Privileged Accesses across Scripts.

Type	Num Scripts	Read		Write		Execute	
		Mean	Median	Mean	Median	Mean	Median
Cross-Owner	343981	4520.63	12	126.36	2	247.18	0
Priv-0 Element	122095	7292.36	6	108.74	3	296.87	0
Priv-1 Element	3023	1243.62	9	6.46	0	82.63	0
Priv-2 Element	181	640.60	21	0.44	0	25.70	0

Priv-0 scripts as children of first-party elements initiated by other Priv-0 scripts. Note that an access to an object in the same privilege level might be determined as cross-owner access if the accessing script is not the initiator of the object. Some people might think such an access is not an abuse of one's privilege. Thus, we demonstrate the results of over-privileged DOM accesses next.

We present in Table 5.8 the mean and median numbers of over-privileged DOM accesses to Priv-0 DOM elements by scripts in the *three* low privilege levels across 82,327 websites. First party scripts in Priv-0 are granted full privilege and hence can access any content they wish. Compared with Table 5.7, the numbers of the three low privileged scripts are close. This result indicates that many cross-owner DOM accesses by Priv-[1-3] scripts are actually over-privileged accesses. In other words, low privileged scripts did not access much content created by other scripts in the same privilege level. Priv-3 third-party scripts, in particular, "abused" their privilege. We enclose the word *abused* in quotation marks because those over-privileged accesses might be for legit purposes. We will inspect whether an cross-owner or over-privileged access is really security sensitive in §5.4.4.

Finally, for each script we compute the average number of cross-owner and over-privileged accesses it made on one web page. The mean and median numbers of cross-owner as well as over-privileged DOM accesses across scripts are demonstrated in Table 5.9. Since there were quite few Priv-1 and Priv-2 elements, we did not find many scripts in a

Table 5.10: Top-10 Scripts by Average Number of Cross-Owner Accesses.

Rank	Script	Num. (%) Websites	Avg. Num. Accesses
1	https://ws.sharethis.com/button/async-buttons.js	108 (100%)	159053.77
2	http://w.sharethis.com/button/async-buttons.js	466 (100%)	135391.04
3	http://cdn.viglink.com/api/vglnk.js	149 (100%)	86497.47
4	http://load.sumome.com/	339 (100%)	68623.35
5	https://load.sumome.com/	241 (100%)	41629.46
6	http://s1.adform.net/banners/scripts/adx.js	204 (100%)	34242.25
7	http://cdn.revcontent.com/build/js/rev2.min.js	136 (100%)	26720.16
8	http://widgets.outbrain.com/outbrain.js	310 (100%)	21848.04
9	https://widgets.outbrain.com/outbrain.js	105 (100%)	21416.42
10	http://cdn.doubleverify.com/avs674.js	161 (100%)	16297.48

Table 5.11: Top-10 Over-Privileged Scripts by Average Number of Priv-0 Element Accesses.

Rank	Script	Num. (%) Websites	Avg. Num. Accesses
1	https://ws.sharethis.com/button/async-buttons.js	108 (100%)	152534.46
2	http://w.sharethis.com/button/async-buttons.js	466 (100%)	129011.62
3	http://cdn.viglink.com/api/vglnk.js	149 (100%)	84487.91
4	http://load.sumome.com/	339 (100%)	63649.95
5	https://load.sumome.com/	241 (100%)	38697.28
6	http://s1.adform.net/banners/scripts/adx.js	146 (72%)	37019.80
7	http://widgets.outbrain.com/outbrain.js	310 (100%)	20348.87
8	https://widgets.outbrain.com/outbrain.js	105 (100%)	19656.70
9	http://cdn.revcontent.com/build/js/rev2.min.js	136 (100%)	17764.19
10	http://cdn.doubleverify.com/avs674.js	161 (100%)	13158.71

lower privilege accessed these objects. There are 182% more scripts that ever performed a cross-owner access than low-privileged scripts that accessed a Priv-0 element. Indeed, many of the cross-owner accessing scripts are first-party scripts. We find that the mean values are significantly greater than the median values, which indicates that quite few scripts performed extensive cross-owner and/or over-privileged accesses. The top-10 cross-owner accessing scripts and top-10 over-privileged scripts that accessed Priv-0 elements are listed in Table 5.10 and Table 5.11, respectively. We sort the scripts by the average number of accesses to Priv-0 elements. Scripts that were found in less than 100 websites are filtered. The numbers in parentheses in the third column are the percentage of websites that one script is included and made cross-owner or over-privileged access. We find that except for http://s1.adform.net/banners/scripts/adx.js, all other 9 scripts performed an cross-owner or over-privileged access on all websites they are embedded in. Surprisingly, the top-10 scripts in Table 5.6 are also the top-10 cross-owner accessing and over-privileged scripts in these two tables. Furthermore, the majority of accesses they did are cross-owner and over-privileged accesses. We investigate whether their accesses are sensitive or not next.

5.4.4 Sensitive DOM API Accesses

We have shown that many third-party Priv-3 scripts were accessing (high-privileged) content created by other scripts. Third-party scripts have legit reasons for accessing others' content. For example, an ad library needs to insert new elements into dedicated region in the DOM tree; a social widget library needs to insert its buttons into specific place on a web page; a developer may use libraries like jQuery to manipulate the DOM. There are also reasons for not letting third-party scripts arbitrarily perform any operation on any content in a web page. First, a lot of sensitive user data (*e.g.*, name, email, address *etc.*) is directly presented in many websites today as an effort to provide personalized experience. Third-party scripts shall not access any of such data. Second, HTML elements such as <form> and <input> are used for collecting user inputs by web applications. Third-party scripts shall not access these tags without a good excuse, either. In particular, modern web browsers automatically fill the saved user login credentials, contact information, and billing information for a user in such elements. All such data is stored in plain text in the DOM tree and can be easily manipulated by JavaScript. Third, sensitive user credentials are usually stored as Cookies in the browser. While an analytic script may need to set a Cookie in the website it is included to track visitors, other third-party scripts have no good reason to access Cookies.

In this section, we investigate whether third-party scripts were abusing their privileges to access sensitive content in a web page. One may argue that without accessing some data first, a script cannot determine whether the accessed data is sensitive or not. Indeed, how to present sensitive user data in a web application depends on the web developer. Thus, we conservatively determine accesses to <form> and <input> elements and to document Cookies as sensitive accesses. Since the global <document> object is not a DOM element, it has no initiator. We manually assign `0` as its privilege. Furthermore, we only study sensitive accesses that are an over-privileged access, as defined in §5.4.3.

Table 5.12: Numbers of Over-Privileged Accesses to Priv-0 Sensitive Elements across 35961 Websites.

Privilege	Read		Write		Execute	
	Mean	Median	Mean	Median	Mean	Median
1	31.77	0	0.90	0	2.06	0
2	1.38	0	0.03	0	0.06	0
3	232.57	30	3.76	0	18.20	0

Accesses to Sensitive Elements

We list the mean and median numbers of over-privileged accesses to Priv-0 sensitive elements across 35,961 websites in Table 5.12. We find that sensitive DOM elements were primarily Read by JavaScript instead of being Written or Executed. Priv-3 scripts still performed a large number of Read accesses on average to Priv-0 sensitive elements. The medians are much smaller than the means, indicating that the majority of sensitive accesses took place on very few websites. On 10% of the 35,961 websites, the Priv-0 sensitive elements were Read, Written and Executed by Priv-3 scripts for at least 376, 8 and 22 times on average, respectively.

For each script we count the numbers of over-privileged sensitive element accesses it made on each web page and then compute the average. We show in Table 5.13 the mean and median numbers of over-privileged sensitive content accesses across scripts. Very few scripts in a lower privilege accessed sensitive elements in Priv-1 or Priv-2. The medians are also much smaller than the means, in particular for Priv-0 sensitive elements. The 99th percentiles of number of Read, Write and Execute accesses to Priv-0 sensitive elements are 6448, 149 and 543, respectively. The top-10 over-privileged scripts that accessed Priv-0 sensitive elements are listed in Table 5.14. Scripts are sorted by the number of websites they had ever accessed one Priv-0 sensitive elements. The numbers in parentheses in the third column are the percentage of websites that one script is included and accessed Priv-0 sensitive elements. Five scripts are served by Facebook and accessed Priv-0 sensitive elements on more than at least 74% of websites they were included. We found that these libraries actually access *all* elements on any web page embedding them. The percentage

Table 5.13: Numbers of Sensitive Content Accesses across Over-Privileged Scripts.

Type	Num Scripts	Read		Write		Execute	
		Mean	Median	Mean	Median	Mean	Median
Cookies	38400	80.73	7	27.27	2		
Priv-0 Element	18192	525.28	25	9.26	0	40.18	1
Priv-1 Element	221	112.08	16	0.59	0	15.32	0
Priv-2 Element	9	36.33	18	0	0	5	0

Table 5.14: Top-10 Over-Privileged Scripts by Number of Websites They Accessed Priv-0 Sensitive Elements.

Rank	Script	Num. (%) Websites	Avg. Num. Accesses
1	https://connect.facebook.net/en_US/sdk.js	4114 (80%)	46.77
2	https://connect.facebook.net/en_US/all.js	2108 (85%)	46.65
3	http://connect.facebook.net/en_US/sdk.js	1040 (83%)	38.50
4	https://mc.yandex.ru/metrika/watch.js	618 (10%)	36.33
5	http://connect.facebook.net/en_US/all.js	540 (88%)	41.44
6	http://cdn.krxd.net/ctjs/controltag.js.7dbac51c9aa7b4135991e8daeb9ced57	530 (80%)	78.75
7	https://connect.facebook.net/ja_JP/sdk.js	497 (74%)	31.67
8	http://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js	471 (6%)	16.05
9	http://pagead2.googlesyndication.com/pagead/js/r20170508/r20170110/show_ads_impl.js	452 (8%)	41.54
10	https://cse.google.com/coop/cse/brand	435 (100%)	12.83

is not 100% because there was no Priv-0 sensitive element on some websites embedding them. Three scripts were served by Google. Two of them are Google's ad libraries and only accessed Priv-0 sensitive elements on less than 8% of the websites including them. However, the other Google's script accessed Priv-0 sensitive elements on every website it was embedded. We find that the string "input" is coded in the rest two scripts from https://mc.yandex.ru and http://cdn.krxd.net, indicating that they intentionally attempted to access <input> tags.

Accesses to Cookies

The mean and median numbers of over-privileged accesses to Cookies across 77,992 websites are demonstrated in Table 5.15. On 10% of the 77,992 websites, their cookies on average were Read and Written by Priv-3 scripts for at least 75and 29 times, respectively. 38,400 Priv-3 scripts had read and written Cookies for 80.73 and 27.27 times on average, respectively. The median number of Read and Write accesses to Cookies are 7 and 2, respectively. The 99th percentiles of number of Read and Write accesses to Cookies are 472 and 112, respectively.

Table 5.15: Numbers of Over-Privileged Accesses to Cookies across 77992 Websites.

Privilege	Read		Write	
	Mean	Median	Mean	Median
1	0.84	0	0.15	0
2	0.06	0	0.02	0
3	38.84	22	13.26	8

Table 5.16: Top-10 Scripts by Number of Websites They Accessed Cookies.

Rank	Script	Num. (%) Websites	Avg. Num. Accesses
1	https://www.google-analytics.com/analytics.js	50520 (100%)	23.05
2	https://www.google-analytics.com/ga.js	10967 (100%)	36.57
3	https://securepubads.g.doubleclick.net/gpt/pubads_impl_117.js	10711 (97%)	12.26
4	https://mc.yandex.ru/metrika/watch.js	5901 (100%)	35.37
5	http://pagead2.googlesyndication.com/pagead/js/r20170508/r20170110/show_ads_impl.js	4696 (85%)	2.57
6	http://pagead2.googlesyndication.com/pagead/js/r20170511/r20170110/show_ads_impl.js	4256 (85%)	2.55
7	https://www.google-analytics.com/plugins/ua/linkid.js	4003 (100%)	1.31
8	https://ssl.google-analytics.com/ga.js	3578 (100%)	34.28
9	https://script.hotjar.com/modules-bcb6f6382be530183b94c4d38f350a82.js	3481 (100%)	3.44
10	https://apis.google.com/_/scs/apps-static/_/js/k=oz.gapi.en_US.PbGgHHLE9-0.O/m=auth/exm=plusone/rt=j/sv=1/d=1/ed=1/am=AQ/rs=AGLTcCNUqA0eeMqhezYrIeahXLBrWBw_dQ/cb=gapi.loaded_1	2759 (100%)	1.00

The top-10 over-privileged scripts that accessed Cookies are listed in Table 5.16. Scripts are sorted by the number of websites they had ever accessed Cookies. Eight scripts are served by Google. Except for the two ad libraries (the 5th and 6th scripts) that accessed Cookies on 85% websites embedding them, the other six scripts accessed Cookies on at least 97% scripts embedding them. In particular, the three Google Analytics scripts and the 10th script loaded from https://apis.google.com accessed Cookies on every page they were included. The rest two scripts from https://mc.yandex.ru and https://script.hotjar.com also accessed Cookies on all websites that they were found. Note that the *watch.js* script from https://mc.yandex.ru is also the 4th script in Table 5.14. We believe Google Analytics and hotjar (one other Analytics service provider) have legitimate reasons to access Cookies. On the other hand, we are concerned with their ability to access other Cookies that belong to the first-party domains. We do not think the other third-party scripts have the need to access the Cookies of the first-party website. Although we do not have evidence showing that their accesses were malicious, such ability can be abused some day. We leave it as a future work.

5.4.5 Summary

We have analyzed the DOM access logs collected with `DOM-Logger` to understand the behaviors of JavaScript code in general. On average most content on a web page is generated by the first party. Third-party scripts made up the majority accesses to DOM elements, including those created by other scripts in a higher privilege. Many third-party scripts, including those served by Google and Facebook, abused their privileges to access sensitive content in web pages. Our results demonstrate the need to restrict the privileges of third-party JavaScript code in today's web applications.

5.5 Related Work

Monitoring JavaScript Behavior. Mostly closed to our monitoring tool is ScriptInspector [130]. It is also a browser-based monitoring tool that can intercept, record and check third-party script accesses to sensitive resources against security policies. Besides the DOM, ScriptInspector can intercept API calls to resources such as local storage and network. However, ScriptInspector is not able to attribute actions to inline scripts that are dynamically generated. Furthermore, it only logs the domain name of the accessing script, thus cannot differentiate distinct scripts loaded from the same domain. As we have discovered, scripts loaded from the same domain and even the same origin can exhibit completely different behaviors. ScriptInspector attributes an access to all unknown third-party domains in the JavaScript call stack, whereas we attribute to the top (last) script in the stack.

Browser extensions like Ghostery [16] can help users understand what third-party script providers are found on the web pages they visit. Browser extensions such as AdBlock Plus [15], uBlock Origin [131] and Privacy Badger [132] can block scripts in specific categories, *e.g.*, tracking and advertising.

Measurement Studies. Yue and Wang presented the first measurement study of insecure JavaScript practices on the web [133] They found more than 66.4% websites included remote scripts and over 44.4% websites used `eval()` to dynamically generate JavaScript

code in a set of 6,805 websites. Ratanaworabhan *et al.* compared the behavior of JavaScript web applications with the JavaScript benchmarks and found that the benchmarks were not representative of many real websites [134]. Richards *et al.* analyzed the dynamic behavior of popular JavaScript libraries [135], and studied the use of `eval()` in popular websites [136]. Lekies *et al.* detected and validated DOM-based XSS vulnerabilities on the Alexa top 5,000 websites [137]. Son and Shmatikov examined insecure use of `postMessage()` and found exploitable vulnerabilities in 84 popular websites [138]. The above work focused on specific unsafe APIs in JavaScript and performed measurement study to detect vulnerable websites. In contrast, we empirically study the use of DOM APIs by JavaScript code and report access to sensitive content in web applications.

Nikiforakis *et al.* did a large-scale crawl of more than three millions web pages from Alexa top 10,000 websites, focusing on the remote JavaScript code inclusion relationships [139]. They analyzed the evolution of JavaScript inclusion over time and developed host-based metrics to assess whether a JavaScript provider could be compromised to serve malicious code. Lauinger *et al.* studied the use of vulnerable or outdated JavaScript libraries over 133K websites and found 37% of the websites included at least one vulnerable library [140]. They leverage the network view of Chrome debugging protocol to determine the causal inclusion relationships. Such approach cannot determine the initiator of dynamically generated inline scripts. Different from these two studies, we focus on how the included remote JavaScript code manipulate content in a web page, rather than how it is included.

5.6 Summary

In this chapter, we have presented `DOM-Logger`, a tool for assisting web developers in monitoring the DOM access behaviors of scripts embedded in their applications. We performed a large-scale crawl of the main pages of the Alexa top 100K websites using `DOM-Logger`. We found that in addition to the 3.13 first-party scripts, 6.8 third-party scripts

were included on a web page on average. The 10 most popular scripts were all served by Google and Facebook. Third-party scripts did not insert much new content into the embedding web pages. Rather, they made extensive accesses to first-party content, including sensitive elements and document Cookies. In particular, we show that some popular scripts from both Google and Facebook were abusing their privileges to access sensitive content on most web pages that embed them. Our results demonstrate the need for more fine-grained approach to restricting the privilege of third-party JavaScript code.

CHAPTER 6
FINE-GRAINED ACCESS CONTROL POLICY FOR THE DOCUMENT OBJECT MODEL

6.1 Motivation

Third-party JavaScript code is commonly embedded in web pages today to enrich user experience. Web developers can easily augment their applications with rich features by including third-party scripts that provide functionalities such as programming enhancement (*e.g.*, jQuery library), visitors tracking (*e.g.*, Google Analytics), advertisements (*e.g.*, DoubleClick), and social integration (*e.g.*, Facebook widget). While some of the libraries can be statically hosted on web developers' own domains, many others are dynamically loaded from the library providers' servers or through Content Delivery Networks (CDN) for auto updating and speeding up page loading [139, 141].

The flexibility of including third-party JavaScript comes with security caveats [139, 109, 111]. Today's web applications handle a wealth of sensitive user data (*e.g.*, password, billing information, email address), which is accessible to all third-party scripts loaded in the same page (origin). Although the Same-Origin Policy (SOP) prohibits accesses from an embedded page (or a parent page) loaded in a different origin, malicious scripts included in the same page (origin) can easily steal such information and eavesdrop on user's input on the page. This is because *they operate at the same privilege level as the owner of the web page*. Furthermore, they can arbitrarily manipulate the page (*e.g.*, modifying content and posting messages on behalf of a user) and include other third-party content, which slow down page rendering and may expose users to malicious links or content (*e.g.*, a drive-by-download attack link).

We observe the *root of cause* of the aforementioned security implications is *the lack of privilege restriction on third-party JavaScript*. Numerous researchers have attempted

to restrict the privileges of third-party content [112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124]. These methods restrict the capabilities and features of JavaScript through 1) rewriting or wrapping third-party JavaScript code using custom APIs [112, 121, 123, 113, 114, 115, 116], or 2) confining third-party scripts in an (isolated) environment to limit their functionality [117, 119, 120, 118, 122, 124]. Many other mechanisms control the flow of sensitive information to prevent information leakage [142, 143, 144, 145, 146, 147, 148]. However, the conventional mechanisms have limitations in terms of *compatibility*, *granularity*, *usability*, and *completeness*. First, some mechanisms completely disable certain features of a third-party script, require JavaScript code rewriting or wrapping, or even modification of server code. Second, they provide coarse-grained protection, *i.e.*, a third-party script is either granted full access to all DOM elements or denied access completely. Third, many solutions request developers to specify security policies for every third-party script that might be included, which is impractical. Lastly, many solutions can only prevent information leakage but not content manipulation.

Content Security Policy (CSP) is a recent web security standard proposed to mitigate the threats from third-part JavaScript code. In particular, CSP prevents cross-site scripting (XSS) and code injection attacks by allowing web developers to designate what content is permitted to be loaded into their web pages. CSP, however, cannot completely solve the above problems, either. First, CSP has not been widely deployed by websites because CSP requires server modification to remove inline JavaScript and `eval` statements [149]. Second, writing a CSP policy without disrupting the application's functionality is difficult and time consuming [150]. Third, the restriction by CSP is still all-or-nothing, *i.e.*, CSP does not restrict the privileges of "malicious" scripts provided by whitelisted and compromised hosts [151, 108].

In this chapter, we focus on the problem of how to restrict both `read` and `write` accesses by third-party JavaScript code at per-object granularity. We have designed DOM Access Control Policy (`DOM-ACP`), a fine-grained and resource-centric access control mechanism to

protect individual DOM elements from unauthorized third-party scripts. DOM-ACP policies are associated directly with resources that need protection, thus are more flexible than capability-based policies. In particular, a rule in a DOM-ACP policy specifies which JavaScript code (*principal*) has what privilege (*access rights*) to access which elements (*resource*). We followed the *fail-safe defaults* and *least privilege* security principles in our design to limit a third-party script's access permission to its necessary minimum, *i.e.*, JavaScript cannot access a protected DOM element unless explicitly granted by the web developer.

DOM-ACP allows developers to write both inline policy and external policy to facilitate easy deployment. We let developers use the new `accessControl` DOM attribute to write access control policy inline for a specific DOM element and its child elements. We also enable developers to use Cascading Style Sheets (CSS) that they are familiar with to define policies for a group of objects in separate policy files, that can be reused across many web pages. The external DOM-ACP policy can be configured with the new *Access-Control-Policy* HTTP header and requires no modification of both server-side and client-side code. DOM-ACP is backward-compatible and does not conflict with existing security mechanisms such as SOP and CSP. Indeed, DOM-ACP works as an additional layer of protection that complements CSP to further restrict the privileges of allowed third-party JavaScript code.

DOM-ACP provides both *confidentiality* and *integrity* to sensitive content in web applications, and significantly reduces the harm that can be caused by malicious third-party scripts. DOM-ACP would not cause compatibility issues for benign third-party scripts because they do not perform unauthorized access to protected sensitive content. Although the current design of DOM-ACP only supports restriction on DOM object access and selecting elements with CSS, DOM-ACP can be easily extended to limit access to other website data (*e.g.*, Cookies, HTML5 Web Storage, and remote communication functions) by third-party JavaScript code, and support XPath selection.

6.2 Background

Preventing unauthorized access to components in web applications[1] is important. In this section, we briefly review two standard security mechanisms that are implemented in today's browsers. We then explain their limitations with two example web applications, motivating us to design DOM-ACP.

6.2.1 Web Permission Policies

Same-Origin Policy (SOP). The SOP is the fundamental concept in today's web application security model. It restricts the interaction among components from different origins (authors), which are 3-tuple of URI scheme, hostname, and port. The policy specifies that a web browser permits scripts in one browsing context [2] to access resources in another browsing context through DOM APIs only if they are from the same origin. For example, resources in the http://a.com context can only be accessed by code executing in other contexts that are in the same origin. Code executing in http://b.com context should certainly never be allowed access.

The SOP prohibits code from a third-party origin from stealing or modifying data from the first-party origin. However, it does not prevent an included third-party JavaScript executing in the browsing context of the first-party origin from disclosing or modifying the data. For example, data can be leaked by writing in the src attribute of an <img> or <script> element or by sending through an XHR request to a remote host.

Content Security Policy. To prohibit or limit communications that are allowed by the SOP, modern browsers have implemented the Content Security Policy (CSP). CSP allows a web developer to create a whitelist of trusted origins through the Content-Security-Policy HTTP header, and instruct the browser to only communicate with and load resources in specific types (*e.g.*, JavaScript, stylesheets, images, *etc.*) from those origins in a browsing context.

[1] Throughout the chapter, we use *JavaScript* and *script* interchangeably.

[2] The different components in a web browser are organized in browsing contexts, which are the content and JavaScript execution environment in a page or frame.

Unsafe JavaScript features (*e.g.*, inline script and `eval()`) can also be disabled by setting CSP rules.

CSP is introduced to prevent cross-site scripting (XSS), code injection and clickjacking attacks. The restriction put by CSP is all-or-nothing. Once a third-party JavaScript is permitted to execute in the context of the first-party origin, CSP does not further limit its access to the DOM in this context. The script may not be able to send data to arbitrary remote origin any more. It can still, however, manipulate the DOM to compromise the integrity of the web application.

6.2.2 Motivating Examples

We introduce our motivating examples with web applications which suffer from *information theft* and *content manipulation* by included malicious third-party scripts. The status-quo web permission policies can protect neither of the two applications from the attacks. These examples demonstrate the requirement for fine-grained and resource-centric access control mechanism for web applications and motivate our design of `DOM-ACP`.

e-Commerce. `e-Commerce` is an online retail website (*e.g.*, `www.e-Commerce.com`). Like many other similar websites, `e-Commerce` has a checkout page that asks for the name, credit card number, expiration date, zip code and other billing information from its customers. `e-Commerce` includes JavaScript from a third-party analytics library (say, `gstatics.com`) in its web application, including the checkout page. This analytics library can help `e-Commerce` attribute transactions to come up with better business strategies, and help operators of `e-Commerce` set up ad campaigns that target customers who did not complete a checkout. However, it can also leak any sensitive data on `e-Commerce` to other websites.

To limit the risk of including this library, the developers of `e-Commerce` may set a CSP rule to load JavaScript only from `gstatics.com`. Such a rule protects the data of `e-Commerce` from being directly leaked to unknown parties. It does not, however, prevent `gstatics.com` from accessing sensitive data (*e.g.*, the billing information) and further sharing with other

parties remotely.

e-Pay. `e-Pay` is an emerging online payment platform. To simplify the website development, `e-Pay` includes the popular jQuery library. Since jQuery is very likely to be cached by a user's browser already, `e-Pay` decides to fetch jQuery dynamically through a Content Delivery Network (CDN) rather than hosting a static version on their own server. Unfortunately, this gives jQuery full access to the content of `e-Pay`, and makes jQuery a very alluring target for attackers. For example, jQuery hosting website was compromised in 2014 [108]. As a result, an attacker can manipulate the payment amount and even change the recipient of a payment to the attacker's account on `e-Pay`, if a user has downloaded the compromised jQuery library.

Unlike the case of `e-Commerce`, where the confidentiality of the web application is at risk, this example shows that the integrity of a website can be compromised by included third-party JavaScript. The SOP certainly cannot prevent such compromises. CSP cannot help either, unless the web developers do not allow any third-party JavaScript to be loaded from remote servers, which is impractical.

6.3 DOM Access Control Policy

This section presents the DOM Access Control Policy (`DOM-ACP`), a new permission mechanism for DOM. `DOM-ACP` provides a fine-grained access control mechanism (§6.3.1), applying access control policies to each DOM *element* or *sub-tree*. This *resource-centric* design lets developers focus on specifying which elements are sensitive while not worrying about the detailed behaviors of each third-party script. The policy rule and language of `DOM-ACP` follows the standard notion of file system permissions in UNIX-like systems. `DOM-ACP` policies can be written either inline with the resources (§6.3.2) or in external policy files (§6.3.2) to enable flexible adoption without source code modification. We further provide a policy generator (§6.3.3) to assist web developers in generating basic policies for their applications.

6.3.1 Design of DOM Access Control Policy

We first discuss the assumptions and goals we made in designing DOM-ACP (§6.3.1), and introduce the components of a DOM-ACP rule (§6.3.1). We then describe how the access control rules are expressed in the form of access control list (§6.3.1), and finally show how the rules are enforced (§6.3.1).

Assumptions and Goals

We assume that all third-party scripts, including those are allowed in CSP policies, could be malicious or compromised. DOM-ACP is designed as an additional layer of protection to complement CSP, by preventing unauthorized third-party scripts from *directly* reading and writing individual or grouped sensitive DOM objects. A script that is granted access privilege might indirectly disclose sensitive data to one that is not. For example, the script can intentionally store sensitive data in a variable in the global scope to allow any other script to access it. Such collusion attacks can be prevented with information flow control techniques, and are out of the scope of DOM-ACP.

Components

We define the three components of a DOM-ACP rule – *resource*, *principal*, and *access right* – as follows.

Resource. The resource is an object to be secured, which is either a DOM object (element)[3] or a DOM (sub-)tree. A web page consists of various resources in terms of sensitivity, *e.g.*, a log-in box implemented with a <form> tag and a website logo displayed in an <img> element. Existing web permission mechanisms allow each third-party script to either access all resources or no resource, which is too coarse-grained. On the contrary, DOM-ACP allows web developers to define flexible and diverse access control rules for data objects in different sensitivity levels. For simplicity, the current design of DOM-ACP only focuses on DOM

[3] We use *object* and *element* interchangeably.

elements. We can easily support other types of resources that are accessible through DOM APIs (*e.g.*, Cookie and Local Storage) in the near future.

Principal. The principal describes the JavaScript code(s) that a `DOM-ACP` rule applies to. A principal can be a specific URL of a script, or a security origin and a host name pattern that represent a set of scripts from an origin and a domain, respectively. For example, the security origin principal `http://www.a.com` represents all JavaScript codes loaded from host `www.a.com` through HTTP and port 80. In our current design, browser extensions are treated as trusted user JavaScript code. As a future work, we plan to include browser extension as principals to restrict their privileges as well.

Access Right. The access right specifies the permitted operations – `read` and/or `write` – of a principal on a resource. These operations can be performed *directly* or *indirectly*. First, the attributes or methods of a DOM object can be directly accessed or called in a JavaScript code. For example, `element.getAttribute("className")` and `var text = element.textContent` directly `read` the `className` and the `textContent` attributes of the element, respectively. Second, a DOM object can be passed as an argument in a receiver DOM object's method, which then performs `read` or `write` operations on the argument internally. For example, `Canvas drawImage()` in HTML5 takes an `Image`, `Video` or `Canvas` object as an argument, reads the object internally and draws it onto a canvas[4]. Furthermore, `DOM-ACP` requires a script to have both `read` and `write` permissions in order to register event handlers on elements.

DOM Access Control List

All `DOM-ACP` rules that apply to the same DOM object are managed in the `DOM access control list (DACL)` data structure, which is associated with the DOM object. Specifically, a `DACL` is a list of access control entries (ACE). Each ACE defines the access right of a principal on the associated object. Every `DACL` has a special ACE, the *default ACE*,

[4] http://www.w3schools.com/tags/canvas_drawimage.asp

which has an empty or the `"default"` principal, and defines the default access permission of all third-party scripts on the corresponding DOM object. A negative default access right (`"None"`) is automatically implied if one is not explicitly defined by the developer. Such *fail-safe defaults* design choice effectively blocks any unauthorized accesses to resources, thus restricting the privileges of third-party scripts to their necessary minimum (*least privilege*).

If a DOM object has child objects, *i.e.*, it is the root node of a DOM sub-tree, its `DACL` is automatically inherited by child objects who do not have one, such that web developers need to define only one policy at the root node and can apply it to all objects in the tree. In case that a DOM object or DOM sub-tree cannot find a `DACL` to inherit, a `DACL` that grants full-privilege (`"RW"`) to all scripts is assigned. Such design is backward-compatible, and eliminates the work of writing `DOM-ACP` policies for resources that do not need protection.

Enforcing Access Control Policy

`DOM-ACP` follows the following rules to determine whether an access should be granted or not.

Accessing Receiver Objects. `DOM-ACP` checks the access right of a script when it accesses a protected DOM object. `DOM-ACP` immediately grants the access if the accessing script is first-party. Otherwise, `DOM-ACP` searches the ACE whose principal best matches the accessing script (`"default"` matches all third-party script) in the receiver object's DACL. The type of access (`read` and/or `write`) is determined by the specific DOM API that is invoked, and is checked with the access right assigned in the ACE. The access is allowed only if all the requested permissions are granted by the developer.

`DOM-ACP` may also need to perform additional permission check in two special cases, where other DOM objects are implicitly accessed in addition to the receiver object.

Accessing Child Objects. Some DOM interfaces provide JavaScript with the ability to access the child objects of a receiver object. For example, invoking the `innerText` property would recursively retrieve the text content of all descendants of a receiver object. When such

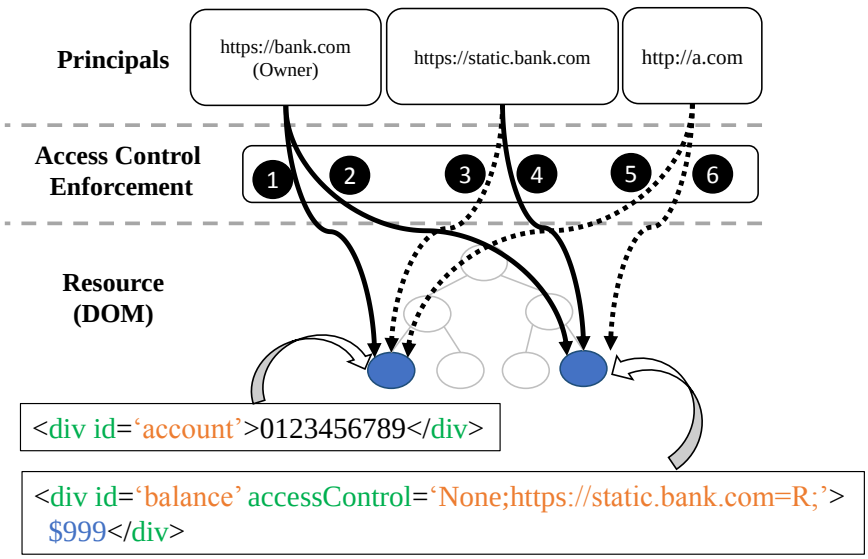


Figure 6.1: Overview of DOM-ACP's design.

attributes or methods are invoked, DOM-ACP also checks the accessing script's permission on all child objects of the receiver object. The access is allowed only if the principal is allowed to access all child objects and the receiver object.

Accessing Arguments of Method. If a DOM method needs to access DOM object(s) that is passed as argument(s), DOM-ACP also performs permission check on the argument object(s) in addition to the receiver object.

6.3.2 Writing an Access Control Policy

The flexibility of DACL allows a web developer to define various access control policies for DOM elements or DOM sub-trees. DOM-ACP supports two different types of policies – inline (§6.3.2) and external (§6.3.2).

Inline DOM-ACP

We introduce the new `accessControl` DOM attribute to let developers directly associate a DOM-ACP policy with elements that need protection. The attribute `accessControl` is *immutable* to third-party script, meaning that even a third-party script granted with full privilege on a DOM object cannot change the policy, nor escalate the privilege of other

scripts. A third-party script can, however, set once the `accessControl` attribute for an element that is dynamically created by it. On the contrary, a first-party script has the ability to update the attribute so that it can dynamically grant/revoke permissions to/from any third-party script.

Figure 6.1 presents an example of HTML snippet that defines an inline `DOM-ACP` policy for the element `#balance`. The attribute `accessControl` of `#balance` carries two rules (ACEs) that are separated by semicolon. The first rule sets the default access right to `"None"`, indicating by default any third-party script has no right to access this DOM object. This negative default rule is optional because `DOM-ACP` can automatically create it. The second ACE indicates that principal `http://static.bank.com` is permitted to perform `read` operation. In other words, this inline policy ensures that, only scripts from `https://static.bank.com` – except for scripts from `https://bank.com` that have full privilege on the entire DOM – are permitted with only the right to read the element `#balance`.

External DOM-ACP

The inline policies are fairly simple and can be easily adopted to small websites. However, they might not be very suitable for complex web applications, whose web pages contain thousands of elements that are dynamically constructed from multiple templates and modules, that might be used in other web pages as well. `DOM-ACP` also supports external policies, in which the same rules can be applied to multiple elements, and can be reused across multiple web pages. External `DOM-ACP` borrows the syntax from Cascading Style Sheet (CSS) and enables developers to select DOM elements with the easy-to-use CSS selectors. Specifically, each access control rule in external `DOM-ACP` consists of a CSS selector and a definition block, which contains one or more ACEs that are separated by semicolons.

```
1 input {
2     "default" = R;
3 }
4 input[type=email].auth,
5 input[type=password].auth {
6     "default" = None;
7     "https://static.bank.com" = RW;
8 }
```

Listing 6.1: An example of external DOM-ACP.

Listing 6.1 is an example of external DOM-ACP. The *1st* rule defined in line 1-3 specifies that all third-party scripts can only read but write <input> elements; the *2nd* rule defined in line 4-8 indicates that <input> elements whose type is either email or password and whose class is auth can only be accessed by scripts from https://static.bank.com (and the owner) with full privilege and no other third-party script can access such elements. In case where two or more rules select the same object, the last rule seen in the policy file is applied. For example, rule #2 precedes rule #1 in Listing 6.1. Furthermore, external DOM-ACP has higher priority over inline DOM-ACP.

6.3.3 Policy Generator

The effectiveness of DOM-ACP on protecting web applications depends on the security policies defined by web developers. However, writing a policy may not be trivial, especially for developers who are not security experts. Thus, we present a tool that helps web developers generate a basic policy for their applications.

The policy generator first defines a rule to prevent all third-party scripts from reading sensitive input fields. Specifically, it uses the **autocomplete** attribute introduced in the HTML5 specification[5] to identify elements that may contain sensitive data. Web developers

[5]https://html.spec.whatwg.org/multipage/forms.html#autofill

can easily extend the rule to cover all elements that may be used for collecting user inputs. They can also add additional rules to protect custom data or grant permission to specific third-party scripts.

Secondly, we leverage our monitoring tool `DOM-Logger` to report all write accesses made by third-party scripts. For each *first-party* element that is modified by a third-party script, we grant the write permission to the accessing script on the element. We use the `css-selector-generator` [6] library to automatically generate an unique CSS selector for an element that is associated with such a rule. Web developers can then revise the policies to disallow or allow certain scripts to write specific nodes. The policy may be very large if too many elements are written in an application. Our policy generator can also combine the rules by gradually removing leaf elements that are children of other elements which are also modified by the same script(s).

6.4 Implementation of DOM-ACP

We implemented a prototype of `DOM-ACP` in Chromium (version 47.0.2526.73). We opt for an implementation in browser because browser can mediate all JavaScript accesses to the DOM, which is analogous to operating system kernel that can mediate all system calls. Figure 6.2 shows the overview of our implementation. Specifically, we extended the *V8 Binding* layer between V8 JavaScript Engine [152] and WebKit [153] in Chromium's rendering engine – Blink [154]. We modified the Blink IDL compiler [155] to insert our access control enforcement code (the gray box in Figure 6.2) into the binding code. The DOM implementation code in WebKit (❸) is not executed if an access is denied by our policy enforcement code (❷).

For instance, in Figure 6.2, the access control policy is enforced as follows: ❶ The JavaScript code tries to access the `id` attribute; ❷ `DOM-ACP` first checks the access control policy, and it proceeds and forwards to WebKit only if it is allowed. Otherwise, it returns an

[6] https://github.com/fczbkk/css-selector-generator

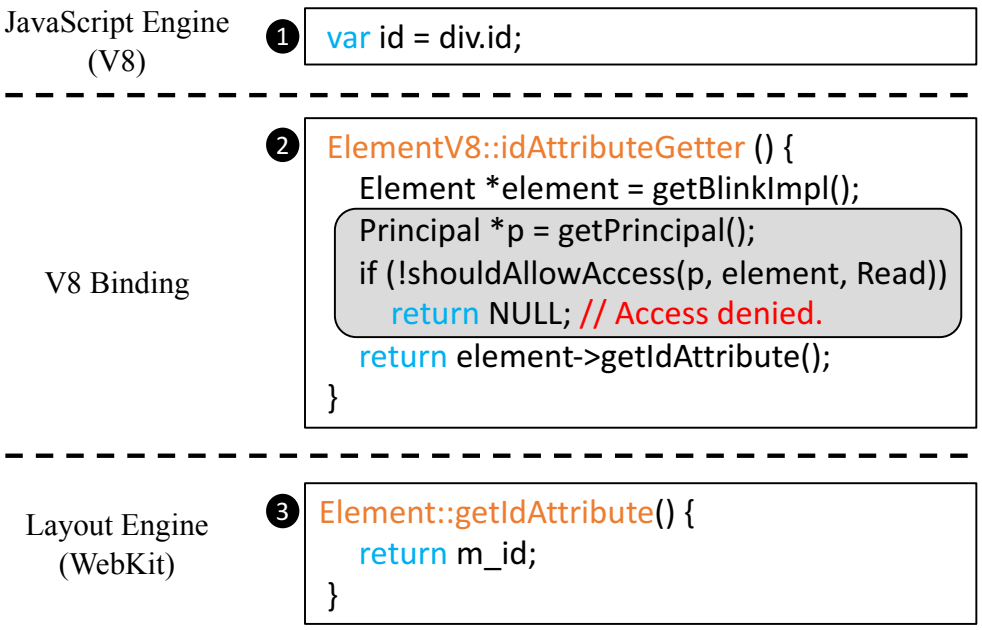


Figure 6.2: Workflow of DOM-ACP's implementation when protecting the id attribute of an element.

error; ❸ WebKit returns the corresponding id as requested.

In the rest of this section, we first briefly introduce V8 Binding in Blink (§6.4.1), then show how inline and external DOM-ACP policies are supported (§6.4.2) and how the policies are enforced in the browser (§6.4.3), and finally discuss how we extended the V8 Binding layer to mediate JavaScript access to DOM elements (§6.4.4).

6.4.1 V8 Binding

In Blink, V8 Binding is the layer between V8 JavaScript engine and WebKit layout engine. Their communication interfaces are specified by Web IDL [156, 157], which is an interface definition language that defines how DOM and other layout engine interfaces are bound to ECMAScript. Blink parses the IDL files and automatically generates the binding code using its IDL compiler when Chromium is being built.

```
1 [ CheckSecurity=Caller
2 ] interface Element : Node {
3     [Reflect] attribute DOMString id;
```

```
4     [CheckSecurity=(Caller,ReadSelf)] DOMString? getAttribute(
          DOMString n);
5 };
```

Listing 6.2: IDL file example.

Listing 6.2 is a simplified IDL file that defines the DOM interface of class `Element`. The keywords within the brackets are *extended attributes*, which are options to control how binding codes are generated inside IDL compiler. Line 3 declares the `id` *attribute* of class `Element`. Figure 6.2 shows how this attribute is implemented in WebKit and exposed to JavaScript. Specifically, WebKit implements the `Element::getIDAttribute()` method in C++ (❸), and exposes this method to V8 through V8 Binding (❷). The binding code in ❷ is automatically generated by the IDL compiler, and is called by V8 when the JavaScript code in ❶ is executed.

6.4.2 Supporting Access Control Policy

Inline Policy

We introduced the **`accessControl`** DOM attribute that developers can use to write inline `DOM-ACP` policy for an element. Specifically, we defined a new read-only attribute in the IDL file of the `Node` interface[7], as **`readonly`** `attribute DOMString` **`accessControl`**, and declared *accessControl* as a new HTML attribute name. The string representation of access control policy specified in the **`accessControl`** attribute is returned when the attribute is accessed in JavaScript. JavaScript cannot directly change the value of this attribute because a setter method was not generated for it by IDL compiler. However, it could be modified indirectly through the `setAttribute("accessControl", "RW")` method or `removeAttribute("accessControl")` method. We addressed the problem by modifying the two methods to disallow third-party scripts to modify the **`accessControl`** attribute of an

[7]Node is the base class of Element and many other classes in DOM.

existing element. Note that a third-party script can set the `accessControl` attribute of an element it creates dynamically, but cannot further modify the value after it is set.

A policy defined in the `accessControl` attribute is transformed to a `DACL`, that can be used to query the access right of a script in Blink. Specifically, the source (URL) of a script is matched with the principal in each ACE and the access permission of the best matching ACE is returned. If the `accessControl` attribute of an `Element` object is not defined, the query is passed to its parent `Element` objects until a policy is found or `NULL` pointer is reached. Such dynamic query allows the automatic propagation of a `DOM-ACP` policy defined at a root element to all its decedent nodes. In the latter case where no policy is defined in any ancestor node, full permission (`"RW"`) is returned to preserve compatibility.

External Policy

We let developers write external `DOM-ACP` policy using the `Access-Control-Policy` HTTP header without modifying any application code. Our current prototype uses a HTTP proxy to parse the external policy and creates the `accessControl` attributes on the fly. As a future work, we plan to discard the proxy workaround, and parse the policy file internally in the browser with CSS parser. We also plan to enable developers to use a <`meta`> tag to include an external `DOM-ACP` policy file just as how they can include an external CSS file today.

6.4.3 Enforcing Access Control Policy

Blink enforces the same-origin policy in class `BindingSecurity`. We extended this class with our access control enforcement code, which is executed in the V8 binding code to check the authority of every access to every DOM object. Figure 6.2 illustrates how `DOM-ACP` enforces a policy.

To check the authority of a JavaScript access, we need to identify the source of JavaScript code, which we implemented in the `getPrincipal()` method (❷). Specifically, we achieved this by looking from the JavaScript Call Stack for the top (oldest) `ScriptCallFrame` object,

that performs the access. The source of the accessing script is returned from the `sourceURL()` method of the `ScriptCallFrame`. In particular, the sources of both inline JavaScript code and bookmarklet are empty in Blink. We instrumented the <script> object creation code inside Blink to track the creation of inline code. If an inline script is created by another JavaScript code, the source of the creator is assigned to the inline script. If it is initially loaded as part of the page HTML source, the source of current document is assigned.

The permission of a script is checked (`shouldAllowAccess()` in ❷) after its source is identified. If the script is first-party, *i.e.*, its security origin is identical to that of the current frame, the access can be immediately granted. Otherwise, we query the `DACL` of the DOM object that is being accessed with the identity of the accessing JavaScript code. The returned permission is compared with the type of access, which can be `Read` and/or `Write` and is determined based on the DOM attribute or method that is being accessed (we will discuss in details later). The DOM implementation in WebKit is called (❸) only if the script has all the required permissions. Otherwise, the access is denied. Note that it is not always necessary to identify the source of a script. An access can be automatically allowed if the `DACL` of the DOM object being accessed has only the *default ACE* that grants full privileges to all JavaScript code. `DOM-ACP` will also check the script's privilege on other objects that can be indirectly accessed in a DOM attribute or method as we discussed in §6.3.1.

6.4.4 Generating Access Control Code

To ensure *complete mediation*, the access control enforcement code implemented in §6.4.3 needs to be executed in every access to every DOM element. We extended the IDL compiler to automatically insert our code into the V8 binding code.

We modified the existing IDL files of all types of DOM elements by using the `CheckSecurity` extended Web IDL attribute. Blink uses this attribute to instruct the IDL compiler to insert security check (*e.g.*, SOP) code into V8 binding code. We introduced the `Caller` and four additional options to this attribute: `ReadSelf`, `WriteSelf`, `AccessChild`, and `WriteChild`.

`Caller` tells the compiler to insert our access control code into the V8 binding code of an attribute or a method of an interface (class). The other four options are used additionally to specify the type of access. `ReadSelf` and `WriteSelf` are used for methods that `read` and `write` the data of the receiver DOM object, respectively. For example, `Element::getAttribute()` can `read`, while `Node::appendChild()` can `write`. In particular, `ReadSelf` and `WriteSelf` are both used for `addEventListener()`. `AccessChild` is used for attributes or methods that additionally access child elements of a receiver object. For instance, `Element::innerHTML` recursively retrieve/modify the content of all child objects in addition to the receiver object. `WriteChild` is similar to `AccessChild` but is used for methods that only modify a child object. `Node::replaceChild(Node node, Node child)` is an example of such methods. Note that `WriteChild` also implies `WriteSelf`. Listing 6.2 is an example of modified IDL file. `CheckSecurity=Caller` is specified at interface level (line 1), meaning that our code will be inserted into the V8 binding code of all members of the interface. `ReadSelf` is additionally used to tell IDL compiler that the method `getAttribute()` performs a `read` operation (line 4).

Depending on the option of `CheckSecurity` and the attribute or method being passed, the IDL compiler inserts the access control code using the following rules.

1. `Attribute`. *Read* and *Write* are the type of access of the *getter* and *setter*[8] methods of an attribute, respectively.

2. `Method`. The four additional attributes indicate the type of access of a method. In particular, the IDL compiler will generate code to perform access check on child objects if `AccessChild` or `WriteChild` is specified. The code calls the `shouldAllowAccess()` method with a child object(s) as argument.

3. `Method accessing other object(s)`. An object can be indirectly accessed as argument of a method, as discussed in §6.3.1. We checked the type of arguments of all methods

[8] A setter method is not generated for read-only attributes.

when the Web IDL files are parsed. If the type of an argument is `Element` or its sub-class, we insert code to perform access check on the argument object as well.

6.4.5 Summary

We closely followed the design of DOM-ACP (§6.3) in our implementation on Chromium. In total, we introduced 1,108 lines of change (1,005 insertions, 103 deletions). Our implementation of DOM-ACP could be easily extended to support any new DOM interfaces in the future by specifying the *CheckSecurity* extended attribute in their IDL files. Although we only implemented a prototype on Chromium (all browsers which use Blink, to be more precisely), the design of DOM-ACP is generic and can be easily implemented on other browser platforms.

6.5 Case Studies

In this section, we demonstrate how DOM-ACP can effectively prevent unauthorized *direct* `read` and/or `write` accesses to protected DOM elements by third-party JavaScript code. In particular, we evaluated DOM-ACP with information theft and content manipulation attacks against popular websites and the two example applications in §6.2.

6.5.1 Protecting Real Websites

In this experiment we aimed to evaluate the effectiveness of DOM-ACP with popular websites and malicious scripts trying to attack them. We visited the Alex US top-100 websites through a HTTP proxy using a vanilla Chromium browser and the DOM-ACP prototype for comparison. We respectively injected two malicious scripts into all web pages with the proxy. First, the information theft script attaches `keypress` event listeners to all <input> DOM elements to steal user-typed authentication data. Second, the content manipulation script removes all child elements of <body> to display a blank web page.

We wrote the external DOM-ACP policies as shown in Listing 6.3 to prevent the attacks

and applied it to all real websites with the proxy. Line 2 and 6 grant full privileges to all scripts so that the real websites can be rendered properly. Line 3 disallows scripts loaded from http://malice-1.net which we used to host the first attack script to access <input> tags. Line 7 allows the second attack script to only `read` but not `write` <body> and its children.

```
1 input {
2     "default" = RW;
3     "http://malice-1.net" = None;
4 }
5 body {
6     "default" = RW;
7     "http://malice-2.net/attack.js" = R;
8 }
```

Listing 6.3: External DOM-ACP policy that protects real websites.

Without surprise, DOM-ACP could prevent the synthetic malicious scripts from attacking the Alexa US top-100 websites whereas the vanilla Chromium browser could not. Specifically, in the vanilla Chromium browser, the information theft script was able to steal authentication data typed by a victim user on 84 out of 90 websites with a log-in feature[9]. In contrast, DOM-ACP prevented the information theft script from accessing any <input> elements. Therefore, the information theft attack was throttled in the very first stage. Similarly, the content manipulation script could delete the content of all the 100 websites in the vanilla Chromium browser. But such attempts were completely prohibited by DOM-ACP.

6.5.2 Protecting e-Commerce and e-Pay

In the second experiment, we protected the two motivating example web applications with DOM-ACP. We used osCommerce[10] as e-Commerce and implemented e-Pay in PHP. We

[9]The attack did not work on 6 websites due to Flash requirement, HTML/JavaScript prompt log-in interface, *etc.*

[10]https://www.oscommerce.com

included a third-party analytics library (https://www.analytics.library) to track user engagement in both applications. We assumed that https://www.analytics.library was compromised. As a result, the compromised script launched two kinds of attacks against the two applications, respectively. The first attack steals billing information entered by a user on `e-Commerce`. The second attack manipulates the recipient of a transfer on `e-Pay`.

We wrote the external `DOM-ACP` policy for `e-Commerce` as shown in Listing 6.4. In particular, the first rule (line 1-4) grants only `read` permission to all third-party scripts and full privilege to the analytics library on <body> and its children. The second rule (line 5-6) prevents any third-party script from accessing elements used for checkout. Specifically, we found that the checkout page templates in `osCommerce` have named their <form> as **name**="checkout_xxx" (line 5). The <input> selector (line 6) was used to automatically select sensitive input elements that can be automatically filled by browsers, such as zip-code (`postal-code`) or credit card number (`cc-number`). The empty definition block implies that `DOM-ACP` should use the negative permission ("None") *default ACE*.

```
1 body {
2     "default" = R;
3     "https://www.analytics.library" = RW;
4 }
5 form[name|=checkout],
6 input[autocomplete] {}
```

Listing 6.4: External `DOM-ACP` policy for `e-Commerce`.

Similarly, we added "transfer-data" to the **class** attribute of the enclosing tag in `e-Pay` and defined two rules. The first rule was identical to the first rule in Listing 6.4 so we omit it in Listing 6.5. The second rule (line 1) denies all third-party scripts' accesses to any elements with **class**="transfer-data" (including their children).

```
1 .transfer-data {}
```

Listing 6.5: Abbreviated external `DOM-ACP` policy for `e-Pay`.

With the two policies, DOM-ACP was able to prevent the above attacks against e-Commerce and e-Pay. The two policies can be used as external DOM-ACP templates for many other similar websites already. The developers of the two websites can also write a CSP policy that blocks the third-party library to prevent the attacks. It, however, comes at the price of loss of functionality.

6.6 Performance Evaluation

In practice, the acceptance of new browser features largely relies upon its performance. In this section, we evaluate the performance overhead of the aforementioned browser prototype in terms of its page load time and peak memory usage.

Setup. We set up our experiment testbed on a Debian Jessie (Linux Kernel 3.16) machine with a quad-core 3.20GHz CPU (Intel Xeon W3565) and 24 GB RAM. We visited the front pages of Alexa US top-100 websites using our prototype and a vanilla build of Chromium (version 47.0.2526.73), and compared page rendering performance between each other. To ensure an accurate measurement, we used both browsers in turn to visit a page 20 times. For each visit, we also reset the user directories for each browser.

Metric. We used *page load time* and *peak memory usage* to evaluate the performance of our prototype system and the vanilla Chromium browser. The page load time represents how long it takes to fetch and render a web page (including third-party content and ads) completely in a browser. The peak memory usage indicates the highest memory usage during a page rendering.

Strategy. The performance of our prototype browser highly depends on number of DOM elements that are protected by a DOM-ACP policy and number of accesses to them by JavaScript. Therefore, we evaluate performance in three different settings: *protecting nothing*, *protecting* <input> *elements* and *protecting everything*. In particular, the first setting uses no DOM-ACP policy, the second setting uses a policy that denies access by third-party scripts to <input> elements, and the last setting uses a policy that denies any

third-party script access. Such policies would break all third-party scripts in real websites. To ensure a fair comparison, we configured DOM-ACP to allow access after the authority check is performed in this experiment.

Protecting nothing. This setting serves as a baseline experiment that measures the performance on visiting legacy websites that do not deploy DOM-ACP. Since none of the elements had the `accessControl` attribute, the accesses were immediately granted by DOM-ACP. As a result, the overheads were negligible: 0% on average in both page load time (median: 0%; standard deviation (σ): 5%) and peak memory usage (median: 0%; σ: 3%).

Protecting input elements. Under the second setting, we observed the overhead in page load time was also negligible (mean and median: 0%; σ: 8%) because very few resources were needed for authority check. The overhead in memory usage also remained negligible (mean and median: 0%; σ: 7%).

Protecting everything. The last setting measures the worst-case performance of the prototype. We discovered that our prototype browser's page load time increased by 276 ms (9.63%) on average across the 100 websites with a median of 82 ms (7%) and a standard deviation of 17% in comparison with the vanilla browser. Our prototype browser did not incur much overhead in memory usage. The memory overhead only increased by approximately 2% on average (median: 1%; σ: 7%).

Summary. Our evaluation showed that the prototype of DOM-ACP introduced modest performance overhead (9% in page load time and 1% in peak memory usage) in the worst case. In real applications there are not too many elements that contain sensitive data, and the overhead would be much lower when securing them only.

6.7 Related Work

In this section, we summarize various techniques related to our DOM protection mechanism.

Protecting DOM elements. DOM-ACP is inspired by several prior researches of protecting sensitive DOM elements. ESCUDO [158] places different elements in rings with decreasing

privileges to prevent XSS attacks. Objects can be accessed only by first-party JavaScript from same or higher privilege rings. DOM-ACP policies are more flexible and easier to deploy than ESCUDO's hierarchical policies. Proctor [159] prevents browser extensions from accessing sensitive DOM elements in a coarse granularity. DOM-ACP can be easily extended to enforce fine-grained access control on extensions. Ryck *et al.* [160] isolate sensitive data in shadow DOM tress from JavaScript code (including first-party) outside the shadow DOM trees. This approach requires significant modification of web applications. In contrast, DOM-ACP does not engage web application re-engineering. ScriptInspector [130] inspects and records third-party script accesses to sensitive elements and is a great tool that can help web developers write DOM-ACP policies.

Constraining JavaScript. Another line of work restricts the functionality of third-party JavaScript to prevent DOM access. ConScript [117], WebJail [118] and TreeHouse [124] support policies that restrict features of third-party script at script or function level. Such restrictions are too coarse-grained, *i.e.*, they cannot selectively protect a subset of DOM elements. JCShadow [122] confines JavaScript objects in multiple isolated shadow JavaScript contexts to provide access control to objects in JavaScript context. DOM-ACP complements JCShadow to provide fine-grained access control to DOM objects. Similarly, Akhawe *et al.* [119, 120] separate an HTML5 application into isolated unprivileged child iframes but require changing application code. Caja [114], ADsafe [115] and ADsafety [116] restrict third-party code to a subset of JavaScript. BrowserShield [161, 112], Phung *et al.* [113], JSand [123], and JaTE [121] rewrite or wrap JavaScript code to enforce security policies without browser modification. Such code instrumentation cannot provide fine-grained access control to DOM elements, either. Furthermore, the above solutions may cause compatibility issues because JavaScript code needs to be rewritten in customized APIs or executed in isolated environment.

Information flow control. DOM-ACP can protect the sensitive data in a website from being directly read by unauthorized scripts. The information flow control (IFC) mechanism

achieves a similar goal that prevents sensitive data from being leaked to untrusted remote servers. Over the past years, a large amount of research utilizes the information flow control technique [142, 143, 144, 145, 146, 147, 148] to identify or prevent information theft attacks in the context of the web. In particular, the IFC technique tracks each piece of sensitive data and analyzes the data flow to ensure the sensitive data does not go across a pre-defined control boundary. While such a technique has been demonstrated to be effective, it introduces high performance overhead in general. For example, the IFC based mechanism in [148] involves a 15.6× overhead. One advantage of IFC solutions is that they can track the data flow among JavaScript objects. Such technique can complement DOM-ACP in preventing privileged scripts colluding with unprivileged ones. On the other hand, IFC technique is less suitable for preventing content manipulation threats that DOM-ACP also aims to mitigate.

6.8 Summary

Although third-party JavaScript code is immensely useful to end users and web application developers, they pose serious security threats due to a lack of fine-grained privilege restriction. In this chapter, we proposed DOM-ACP to protect sensitive data in web applications from being directly accessed by unauthorized third-party JavaScript code. DOM-ACP allows web developers to specify inline and external access control policies for the DOM objects that need protection. DOM-ACP prohibits direct read and write access to the protected DOM objects from unauthorized third-party scripts. DOM-ACP is backward-compatible and does not require modification of application code. We implemented a prototype of DOM-ACP on the popular Chromium browser and demonstrated the effectiveness of DOM-ACP in safe-guarding sensitive data in web applications while introducing modest overhead.

CHAPTER 7
CONCLUSION

Third-party content is widely included by modern web applications and mobile applications to enhance functionality. It usually operates at full privilege as the embedding application and can make unrestricted accesses to all the content, including sensitive direct user data, in the application. A third-party content provider can also gather indirect data about a user in an application and across multiple applications that embed its content, often without the user's consent. In this dissertation, we explored the problem that the confidentiality and integrity of a user's direct data and indirect data may be compromised through the practices of embedding third-party content.

We have identified two classes of new threats targeting individual user arising from embedding third-party content in web applications and mobile applications. Specifically, we demonstrated that a malicious *first-party* application can embed content of other applications to *pollute* and *infer* a user's indirect data in the other applications. These findings advanced the security research community's understanding about the emerging threats that can be posed to end users. We also developed new technique to provide end users with selective control to opt out from unwanted web tracking. Our new technique achieved a good balance between a user's need for privacy and the user's online experience. We further studied the over-privileged third-party JavaScript code in modern web applications and proposed new permission mechanism to restrict the privilege of third-party script. Our study revealed that the over-privileged third-party scripts made extensive accesses to first-party web content, including sensitive content, in today's web applications. A very disconcerting observation is that such over-privileged scripts were prevalent in today's web applications.

Although this dissertation has studied only very few applications and threats, we vision that many more threats caused by the practices of embedding content from third-party

providers will emerge in the future. In particular, powerful *cross-platform* third-party providers can potentially put serious threats to end users. The fundamental causes of these threats are that a lot of system and application designs do not follow the *least-privilege* security principle, nor do they provide *sufficient isolation* between content that mutually distrusts.

Without taking a principled and security-by-design approach to (re)designing existing insecure systems and new systems, we can never prevent new threats from emerging. Admittedly, preserving the flexibility and benefit of embedding content while ensuring security cannot be an easy task. Many mitigation techniques require significant change of existing systems and applications. They suffer from the criticism about harming compatibility or usability and are rarely deployed in practice. On the other hand, techniques that preserve compatibility and good user experience often do not provide sufficient security guarantee in rare cases. We conclude that designing compatible and usable techniques that secure mutually-distrusted content in modern applications remains as an open, important and challenging direction.

REFERENCES

[1] *Ecommerce Chart: The most effective types of personalized product recommendations*, https://www.marketingsherpa.com/article/chart/personalized-product-recommendations, 2015.

[2] A. Farahat and M. C. Bailey, "How effective is targeted advertising?" In *Proceedings of the 21st International Conference on World Wide Web*, ACM, 2012, pp. 111–120.

[3] *The Value of Behavioral Targeting*, http://www.networkadvertising.org/pdfs/Beales_NAI_Study.pdf, 2009.

[4] X. Xing, W. Meng, D. Doozan, A. C. Snoeren, N. Feamster, and W. Lee, "Take This Personally: Pollution Attacks on Personalized Services," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, USA, Aug. 2013.

[5] *Amazon.com product identifiers*, http://archive.org/details/asin_listing.

[6] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, Alexandria, Virginia, USA: ACM, 2008, pp. 75–88.

[7] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Proceedings of the 21st USENIX Conference on Security Symposium*, USENIX Association, 2012, pp. 22–22.

[8] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *In Proceeding of the Network and Distributed System Security Symposium*, 2007.

[9] *AdSense*, www.google.com/adsense/.

[10] B. Liu, A. Sheth, U. Weinsberg, J. Chandrashekar, and R. Govindan, "Adreveal: Improving transparency into online targeted advertising," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ACM, 2013, 12:1–12:7.

[11] K. Springborn and P. Barford, "Impression fraud in online advertising via pay-per-view networks," in *Proceedings of the 22nd USENIX Conference on Security Symposium*, USENIX Association, 2013, pp. 211–226.

[12] P. Gill, V. Erramilli, A. Chaintreau, B. Krishnamurthy, K. Papagiannaki, and P. Rodriguez, "Follow the money: Understanding economics of online aggregation

and advertising," in *Proceedings of the 2013 Conference on Internet Measurement Conference*, ser. IMC '13, Barcelona, Spain: ACM, 2013, pp. 141–148.

[13] *What's Trending in Display for Publishers?* http://www.google.com/think/research-studies/whats-trending-in-display-for-publishers.html.

[14] *Google Ad Preferences Manager*, https://www.google.com/ads/preferences.

[15] *Adblock Plus*, http://adblockplus.org/.

[16] *Ghostery*, http://www.ghostery.com/.

[17] *Google Webmaster Tools - Frames*, https://support.google.com/webmasters/answer/34445?hl=en.

[18] V. Dave, S. Guha, and Y. Zhang, "Viceroi: Catching click-spam in search ad networks," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, Berlin, Germany: ACM, 2013, pp. 765–776.

[19] *Facebook Ads*, https://www.facebook.com/settings?tab=ads&view.

[20] *Microsoft personalized ad preferences*, http://choice.microsoft.com/en-us/opt-out.

[21] *Yahoo Ad Interest Manager*, http://info.yahoo.com/privacy/us/yahoo/opt_out/targeting/details.html.

[22] *Amazon.com: Advertising Preferences*, http://www.amazon.com/gp/dra/info.

[23] B. Wu and B. D. Davison, "Identifying link farm spam pages," in *Special interest tracks and posters of the 14th international conference on World Wide Web*, ser. WWW '05, Chiba, Japan: ACM, 2005, pp. 820–829.

[24] L. Lu, R. Perdisci, and W. Lee, "Surf: Detecting and measuring search poisoning," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11, Chicago, Illinois, USA: ACM, 2011, pp. 467–476.

[25] D. S. Evans, "The Economics of the Online Advertising Industry," *Review of Network Economics*, vol. 7, no. 3, pp. 359–391, 2008.

[26] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 413–427.

[27] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012, pp. 155–168.

[28] V. Anupam, A. Mayer, K. Nissim, B. Pinkas, and M. K. Reiter, "On the security of pay-per-click and other web advertising schemes," in *Proceedings of the Eighth International Conference on World Wide Web*, ser. WWW '99, Toronto, Canada: Elsevier North-Holland, Inc., 1999, pp. 1091–1100.

[29] A. Metwally, D. Agrawal, and A. E. Abbadi, "Using association rules for fraud detection in web advertising networks," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, Trondheim, Norway: VLDB Endowment, 2005, pp. 169–180, ISBN: 1-59593-154-6.

[30] N. Daswani, C. Mysen, V. Rao, S. Weis, K. Gharachorloo, and S. Ghosemajumder, "Online advertising fraud," *Crimeware: understanding new attacks and defenses*, 2008.

[31] B. Stone-Gross, R. Stevens, A. Zarras, R. Kemmerer, C. Kruegel, and G. Vigna, "Understanding fraudulent activities in online ad exchanges," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11, Berlin, Germany: ACM, 2011, pp. 279–294.

[32] V. Dave, S. Guha, and Y. Zhang, "Measuring and fingerprinting click-spam in ad networks," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 175–186, Aug. 2012.

[33] *Google AdSense - Working better together: Protecting against invalid activity*, http://adsense.blogspot.com/2012/12/working-better-together-protecting.html.

[34] G. Wang, C. Wilson, X. Zhao, Y. Zhu, M. Mohanlal, H. Zheng, and B. Y. Zhao, "Serf and turf: Crowdturfing for fun and profit," in *Proceedings of the 21st international conference on World Wide Web*, ser. WWW '12, Lyon, France: ACM, 2012, pp. 679–688.

[35] O. Stitelman, C. Perlich, B. Dalessandro, R. Hook, T. Raeder, and F. Provost, "Using co-visitation networks for detecting large scale online display advertising exchange fraud," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '13, Chicago, Illinois, USA: ACM, 2013, pp. 1240–1248.

[36] S. A. Alrwais, A. Gerber, C. W. Dunn, O. Spatscheck, M. Gupta, and E. Osterweil, "Dissecting ghost clicks: Ad fraud via misdirected human clicks," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, Orlando, Florida: ACM, 2012, pp. 21–30.

[37] L. Zhang and Y. Guan, "Detecting click fraud in pay-per-click streams of online advertising networks," in *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ser. ICDCS '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 77–84, ISBN: 978-0-7695-3172-4.

[38] S. Nath, "Madscope: Characterizing mobile in-app targeted ads," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, ACM, 2015, pp. 59–73.

[39] T. Book and D. S. Wallach, "An empirical study of mobile ad targeting," *arXiv:1502.06577*, 2015.

[40] C. Castelluccia, M.-A. Kaafar, and M.-D. Tran, "Betrayed by your ads!: Reconstructing user profiles from targeted ads," in *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, 2012, pp. 1–17.

[41] L. Olejnik, T. Minh-Dung, C. Castelluccia, *et al.*, "Selling off privacy at auction," in *Proceedings of the 2014 Network and Distributed System Security Symposium*, ISOC, 2014.

[42] *Smartphone os market share, q4 2014*, http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[43] T. Book and D. S. Wallach, "A case of collusion: A study of the interface between ad libraries and their apps," in *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, ACM, 2013, pp. 79–86.

[44] *Targeting - admob android guides - google developers*, https://developers.google.com/mobile-ads-sdk/docs/admob/android/targeting.

[45] *Topic Targeting*, https://support.google.com/partners/answer/2769377?hl=en.

[46] *Reach people interested in your products or services*, https://support.google.com/adwords/answer/2497941?hl=en.

[47] *Reach people of specific demographics*, https://support.google.com/adwords/answer/2580383?hl=en.

[48] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, USENIX Association, 2012, pp. 553–567.

[49] *Ads take a step towards "https everywhere"*, http://adwords.blogspot.com/2015/04/ads-take-step-towards-https-everywhere.html.

[50] *Add demographic targeting to an ad group*, https://support.google.com/adwords/answer/2580282?hl=en.

[51] *Amazon mechanical turk*, https://www.mturk.com/mturk/welcome.

[52] *Google Ads Settings*, http://www.google.com/ads/preferences.

[53] *Phantomjs*, http://phantomjs.org/.

[54] *Yahoo content analysis api*, https://developer.yahoo.com/contentanalysis/.

[55] *Targeting options - advertisers - mobgold*, http://www.mobgold.com/web/advertiser/target-option.

[56] *Adopting encryption: The need for https*, http://www.iab.net/iablog/2015/03/adopting-encryption-the-need-for-https.html.

[57] *Scikit-learn: Machine learning in python*, http://scikit-learn.org/stable/.

[58] *Overview of race and hispanic origin: 2010*, http://www.census.gov/prod/cen2010/briefs/c2010br-02.pdf.

[59] *Privacy policy - privacy & terms - google*, http://www.google.com/policies/privacy/.

[60] *Key terms- privacy & terms - google*, http://www.google.com/policies/privacy/key-terms/#toc-terms-sensitive-categories.

[61] F. T. Commission, "Data brokers: A call for transparency and accountability," 2014.

[62] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ACM, 2012, pp. 71–72.

[63] E. Balsa, C. Troncoso, and C. Diaz, "Ob-pws: Obfuscation-based private web search," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2012, pp. 491–505.

[64] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 674–689.

[65] M. Lécuyer, G. Ducoffe, F. Lan, A. Papancea, T. Petsios, R. Spahn, A. Chaintreau, and R. Geambasu, "Xray: Enhancing the web's transparency with differential cor-

relation," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX Association, 2014, pp. 49–64.

[66] A. Korolova, "Privacy violations using microtargeted ads: A case study," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, 2010, pp. 474–482.

[67] P. Barford, I. Canadi, D. Krushevskaja, Q. Ma, and S. Muthukrishnan, "Adscape: Harvesting and analyzing online display ads," in *Proceedings of the 23rd International Conference on World Wide Web*, ACM, 2014, pp. 597–608.

[68] A. Datta, M. C. Tschantz, and A. Datta, "Automated experiments on ad privacy settings," *Proceedings on Privacy Enhancing Technologies*, vol. 1, no. 1, pp. 92–112, 2015.

[69] S. Guha, B. Cheng, and P. Francis, "Privad: Practical privacy in online advertising," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, 2011, pp. 169–182.

[70] A. Reznichenko and P. Francis, "Private-by-design advertising meets the real world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 116–128.

[71] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas, "Adnostic: Privacy preserving targeted advertising," in *Proceedings of the 2010 Network and Distributed System Security Symposium*, ISOC, 2010.

[72] M. Fredrikson and B. Livshits, "Repriv: Re-envisioning in-browser privacy," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2011, pp. 131–146.

[73] M. Backes, A. Kate, M. Maffei, and K. Pecina, "Obliviad: Provably secure and practical online behavioral advertising," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2012, pp. 257–271.

[74] N. Mor, O. Riva, S. Nath, and J. Kubiatowicz, "Bloom cookies: Web search personalization without user tracking," in *Proceedings of the 2015 Network and Distributed System Security Symposium*, ISOC, 2015.

[75] M. Hardt and S. Nath, "Privacy-aware personalization for mobile advertising," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM, 2012, pp. 662–673.

[76] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *Proceedings of the 22nd USENIX Conference on Security Symposium*, USENIX Association, 2013, pp. 97–112.

[77] P. Eckersley, *Stop sneaky online tracking with eff's privacy badger*, https://www.eff.org/press/releases/stop-sneaky-online-tracking-effs-privacy-badger.

[78] J. Jao, *Perils of personalization vs. privacy*, http://www.mediapost.com/publications/article/210880/perils-of-personalization-vs-privacy.html?edition=.

[79] R. K. Chellappa and R. G. Sin, "Personalization versus privacy: An empirical examination of the online consumer's dilemma," *Inf. Technol. and Management*, vol. 6, no. 2-3, pp. 181–202, Apr. 2005.

[80] Uber, *User privacy statement*, https://www.uber.com/legal/privacy/users/en.

[81] Google, *Google ads preferences manager*, https://www.google.com/settings/ads/onweb.

[82] Yahoo, *Ad interest manager - yahoo*, https://info.yahoo.com/privacy/us/yahoo/opt_out/targeting/.

[83] R. Calo, *Does nai's opt out tool stop consumer tracking?* http://cyberlaw.stanford.edu/blog/2009/04/does-nai%E2%80%99s-opt-out-tool-stop-consumer-tracking.

[84] P. Dixon, *THE NETWORK ADVERTISING INITIATIVE: Failing at Consumer Protection and at Self-Regulation*, http://www.worldprivacyforum.org/wp-content/uploads/2007/11/WPF_NAI_report_Nov2_2007fs.pdf.

[85] S. Garfinkel, *Database Nation: The Death of Privacy in the 21st Century*. O'Reilly & Associates, Inc., 2000.

[86] *Browser fingerprints, and why they are so hard to erase*, http://www.networkworld.com/article/2884026/security0/browser-fingerprints-and-why-they-are-so-hard-to-erase.html.

[87] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *Proceedings of the 24th International Conference on World Wide Web*, Florence, Italy, 2015, pp. 820–830.

[88] *Tor browser*, https://www.torproject.org/projects/torbrowser.html.en.

[89] *Multi-process architecture - the Chromium projects*, https://www.chromium.org/developers/design-documents/multi-process-architecture.

[90] *Preferences in the chromium projects*, https://www.chromium.org/developers/design-documents/preferences.

[91] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with wprof," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, Lombard, IL, 2013, pp. 473–486.

[92] *The xml c parser and toolkit of gnome*, http://www.xmlsoft.org/.

[93] F. Sun, D. Song, and L. Liao, "Dom based content extraction via text density," in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, Beijing, China, 2011, pp. 245–254.

[94] X. Qi and B. D. Davison, "Web page classification: Features and algorithms," *ACM Comput. Surv.*, vol. 41, no. 2, 12:1–12:31, Feb. 2009.

[95] G. Pass, A. Chowdhury, and C. Torgeson, "A picture of search," in *Proceedings of the 1st International Conference on Scalable Information Systems*, Hong Kong, 2006.

[96] *Taxonomy api | alchemyapi*, http://www.alchemyapi.com/products/alchemylanguage/taxonomy.

[97] *Taxonomy api documentation | alchemyapi*, http://www.alchemyapi.com/api/taxonomy.

[98] *abine*, https://www.abine.com/index.html.

[99] *Network Advertising Initiative (NAI)*, https://www.networkadvertising.org/.

[100] *Digital Advertising Alliance (DAA)*, http://www.aboutads.info/.

[101] *Browser Fingerprints: A Big Privacy Threat*, http://www.pcworld.com/article/192648/browser_fingerprints.html.

[102] *Lawsuit: Ad Network Could Be Tracking You With HTML5*, http://www.techhive.com/article/205950/ad_network_abuses_html5_privacy_advocates_cry_foul.html.

[103] *Local Shared Objects – "Flash Cookies"*, https://epic.org/privacy/cookies/flash.html.

[104] *Browse in private with incognito mode*, https://support.google.com/chrome/answer/95464?hl=en.

[105] X. Pan, Y. Cao, and Y. Chen, "I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser," in *Proceedings of the 2015 Network and Distributed System Security Symposium*, 2015.

[106] *Cross-browser fingerprinting test 2.0*, http://fingerprint.pet-portal.eu/?menu=6.

[107] K. Boda, A. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, Tallinn, Estonia, 2012, pp. 31–46.

[108] R. Whitbeck, *Update on jquery.com compromises*, https://blog.jquery.com/2014/09/24/update-on-jquery-com-compromises/, Sep. 2014.

[109] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proceedings of the 19th International World Wide Web Conference (WWW)*, 2010.

[110] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in javascript web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Oct. 2010.

[111] Z. Li, S. Alrwais, X. Wang, and E. Alowaisheq, "Hunting the red fox online: Understanding and detection of mass redirect-script injections," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[112] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," *ACM Trans. Web*, vol. 1, no. 3, Sep. 2007.

[113] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting JavaScript," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)*, 2009.

[114] *Caja*, https://developers.google.com/caja/.

[115] *Adsafe*, http://www.adsafe.org/.

[116] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: Type-based verification of javascript sandboxing," in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[117] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[118] S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen, "WebJail: Least-privilege integration of third-party components in web mashups," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[119] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in HTML5 applications," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[120] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song, "Data-confined HTML5 applications," in *Proceedings of the 18th European Symposium on Research in Computer Security*, Egham, UK, Sep. 2013.

[121] T. Tran, R. Pelizzi, and R. Sekar, "JaTE: Transparent and efficient JavaScript confinement," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[122] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, "Towards fine-grained access control in JavaScript contexts," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[123] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[124] L. Ingram and M. Walfish, "TreeHouse: JavaScript sandboxes to help web developers help themselves," in *Proceedings of the 2012 ATC Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.

[125] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "Dlint: Dynamically checking bad coding practices in javascript," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, Baltimore, MD, USA, 2015.

[126] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, Russia, 2013.

[127] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[128] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015.

[129] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th International World Wide Web Conference (WWW)*, Perth, Australia, Apr. 2017.

[130] Y. Zhou and D. Evans, "Understanding and monitoring embedded web scripts," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[131] *Ublock origin*, https://github.com/gorhill/uBlock.

[132] *Privacy badger*, https://www.eff.org/privacybadger.

[133] C. Yue and H. Wang, "Characterizing insecure javascript practices on the web," in *WWW*, 2009.

[134] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications," in *Proceedings of the 2010 USENIX Conference on Web Application Development*, Boston, MA, 2010.

[135] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, 2010.

[136] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *Proceedings of the 25th European Conference on Object-oriented Programming*, Lancaster, UK, 2011.

[137] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[138] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in html5 websites," in *Proceedings of the 2013 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[139] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote JavaScript inclusions," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[140] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2016.

[141] S. Arshad, A. Kharraz, and W. Robertson, "Include me out: In-browser detection of malicious third-party content inclusions," in *Proceedings of the 20th International Conference on Financial Cryptography and Data Security (FC)*, Barbados, Feb. 2016.

[142] A. Yip, N. Narula, M. Krohn, and R. Morris, "Privacy-preserving browser-side scripting with BFlow," in *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, Mar. 2009.

[143] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "FlowFox: A web browser with flexible and precise information flow control," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[144] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian, "Run-time monitoring and formal analysis of information flows in Chromium," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[145] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, and D. Heman, "Protecting users by confining JavaScript with COWL," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[146] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in JavaScript and its APIs," in *Proceedings of the 29th Symposium on Applied Computing (SAC)*, 2014.

[147] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer, "Information flow control for event handling and the DOM in web browsers," in *Proceedings of the 28th IEEE Computer Security Foundataions Symposium (CSF)*, 2015.

[148] A. Chudnov and D. A. Naumann, "Inlined information flow monitoring for JavaScript," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[149] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[150] M. Fazzini, P. Saxena, and A. Orso, "AutoCSP: Automatically retrofitting CSP to web applications," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 2015.

[151] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[152] *Chrome V8 - Google developers*, https://developers.google.com/v8/.

[153] *WebKit*, https://webkit.org/.

[154] *Blink - the Chromium projects*, https://www.chromium.org/blink/.

[155] *IDL compiler - the Chromium projects*, https://www.chromium.org/developers/design-documents/idl-compiler.

[156] *Web IDL (second edition)*, https://heycam.github.io/webidl/.

[157] *Web IDL in Blink - the Chromium projects*, https://www.chromium.org/blink/webidl.

[158] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin, "ESCUDO: A fine-grained protection model for web browsers," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2010.

[159] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome extensions: Threat analysis and countermeasures," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.

[160] P. De Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen, "Protected web components: Hiding sensitive information in the shadows," *IT Professional*, vol. 17, no. 1, pp. 36–43, Jan. 2015.

[161] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.